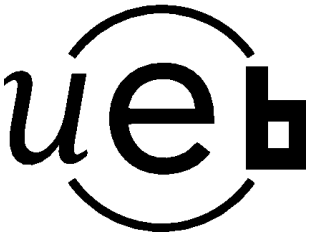
Gestion de la variabilité et automatisation des processus de développement logiciel

Emmanuelle Rouillé

ANNÉE 2014



THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

École doctorale Matisse

présentée par

Emmanuelle ROUILLÉ

préparée à l'unité de recherche IRISA – UMR6074
Institut de Recherche en Informatique et Systèmes Aléatoires

Gestion de la variabilité et automatisation des processus de développement logiciel

Thèse soutenue à Rennes
le 16 avril 2014

devant le jury composé de :

Marie-Pierre GERVAIS
Professeur à l'Université de Paris Ouest Nanterre La Défense / *Présidente*

Pierre-Alain MULLER
Professeur à l'Université de Haute-Alsace / *Rapporteur*

Bernard COULETTE
Professeur à l'Université de Toulouse II - Le Mirail / *Rapporteur*

Reda BENDRAOU
Maître de Conférence à l'Université Pierre et Marie Curie / *Examinateur*

Benoit COMBEMALE
Maître de Conférence à l'Université de Rennes 1 / *Examinateur*

Olivier BARAIS
Maître de Conférence à l'Université de Rennes 1 / *Examinateur*

David TOUZET
Architecte logiciel chez Sodifrance / *Examinateur*

Jean-Marc JÉZÉQUEL
Professeur à l'Université de Rennes 1 / *Directeur de thèse*

Remerciements

Je remercie le Professeur Bernard Coulette et le Professeur Pierre-Alain Muller d'avoir rapporté ce manuscrit, ainsi que pour leurs retours ayant permis de l'améliorer. Je les remercie également d'avoir fait partie de mon jury de thèse. Je remercie le Professeur Marie-Pierre Gervais d'avoir accepté de présider à ma soutenance, et je remercie le Docteur Reda Bendraou d'avoir examiné mes travaux. Merci à Jean-Marc Jézéquel d'avoir dirigé et également permis cette thèse. Un grand merci à Benoît Combemale, Olivier Barais, David Touzet, et à travers ce dernier Sodifrance, pour m'avoir encadrée, pour leur disponibilité, leur investissement et les nombreux conseils qu'ils m'ont prodigués. Merci à tous les collègues de DiverSE et de Sodifrance pour les discussions intéressantes et tous les bons moments partagés ensemble. Enfin, merci à ma famille, Clément, mes parents, pour leur présence et leur soutien.

Résumé

De nombreux outils ont été proposés par la communauté du génie logiciel afin de faire face à la complexité des logiciels et des projets de développement logiciel. À titre d'illustration, les systèmes de contrôle de version permettent de gérer les difficultés liées au travail collaboratif et géographiquement distribué, les outils de compilation, de test ou d'intégration continue supportent les méthodes de développement agiles, les IDE (*Integrated Development Environments*) permettent d'intégrer les différentes technologies utilisées sur un projet, etc. Ces outils sont d'autant plus nombreux qu'il en existent des version différentes, afin de satisfaire les besoins spécifiques à chaque projet. Par exemple des IDE différents peuvent être utilisés en fonction du langage de programmation utilisé. L'utilisation de tous ces outils est cependant à l'origine de nombreuses tâches manuelles répétitives, sources d'erreurs et coûteuses en temps.

L'automatisation de ces tâches est un moyen de gagner en productivité. Mais la difficulté est de réutiliser des automatisations de tâches manuelles répétitives. En effet, toutes les automatisations ne sont pas forcément utiles pour tous les projets et elles peuvent avoir des dépendances avec d'autres tâches réalisées pendant un projet. De plus, une automatisation peut être utilisée dans des cas différents, c'est-à-dire dans des projets différents ou à des moments différents d'un même projet. Les défis sont donc de déterminer les projets et les moments d'un projet pour lesquels une automatisation doit être réutilisée, ainsi que de créer des automatisations réutilisables à travers leurs différents cas d'utilisation.

Afin de répondre à ces défis, la contribution principale de cette thèse est une approche pilotant la réutilisation des automatisations de tâches manuelles répétitives par les processus de développement logiciel, où un processus de développement logiciel décrit les étapes à réaliser pour mener à bien un projet de développement logiciel. Cette approche consiste à capitaliser sur un ensemble de processus et à réutiliser des processus de cet ensemble en fonction des exigences des projets. Elle s'appuie sur une première sous-contribution, à savoir une approche permettant de réutiliser les processus indépendamment du formalisme utilisé pour les définir. L'approche principale consiste également à lier des automatisations de tâches manuelles répétitives aux étapes d'un ensemble de processus qu'elles automatisent. Ce lien permet de savoir quelles automatisations utiliser pour un projet donné et quand. L'approche principale s'appuie également sur une deuxième sous-contribution, à savoir une méthodologie fournissant un support à la création d'automatisations réutilisables à travers leurs différents cas d'utilisation. Dans cette méthodologie, le lien entre les processus et les automatisations est utilisé afin d'expliciter les différents cas d'utilisation de ces dernières et de créer des automatisations réutilisables.

Afin de démontrer la faisabilité de l'approche proposée dans cette thèse, nous proposons un outillage la supportant. Cet outillage a été appliqué a deux cas d'utilisation : une famille de processus industriels de développement Java ainsi qu'une famille de processus consistant à définir et outiller un langage de modélisation.

Table des matières

Chapitre 1

Introduction

1.1 La prolifération des outils de développement logiciel

La variabilité et l'évolution des exigences d'un projet de développement logiciel sont connues comme une des sources principales de complexité de l'activité de création logiciel [TI93, JMD04, Mar03]. En outre, les projets de développement logiciel sont de plus en plus réalisés de manière collaborative [HMR06], dans un contexte géographiquement distribué [CBHB07], et dans un contexte de globalisation faisant collaborer un ensemble d'entreprises au sein d'un même projet. Ces caractéristiques apportent une complexité accidentelle supplémentaire à la problématique du développement logiciel. Un des objectifs du génie logiciel est de conserver l'intelligibilité du processus de construction et de maintenance de ces applications. Dans ce cadre, de nombreux outils sont apparus afin de gérer, voire automatiser, une partie de cette complexité. Cependant, de part le nombre d'acteurs intervenant dans la construction d'un logiciel et leurs multiples préoccupations, il existe maintenant un écosystème riche et lui-même complexe lié à la problématique de développement logiciel.

Parmi ces outils, sans être exhaustifs, nous pouvons citer :

- des systèmes de contrôle de version (SVN, Git...) qui permettent de gérer les difficultés liées au travail collaboratif et géographiquement distribué,
- des outils de compilation (Maven, Ant, PDE...) qui permettent de gérer les problématiques d'intégration de composants ou de librairies réutilisables au moment de la construction d'un logiciel complexe,
- des outils d'analyse statique comme PMD, Findbugs ou Sonar qui aident à améliorer la qualité du logiciel,
- des outils de support à l'exécution des différents types de test logiciel (JUnit, Selenium, Quality Center...),
- des environnements de développement intégrés (IDE, *Integrated Development Environment*) permettant d'intégrer les différentes technologies utilisées sur un projet (langages, outils, *frameworks*, librairies...),
- des outils d'intégration continue permettant d'automatiser une partie de l'orchestration de la problématique de production logicielle.

La forte variabilité des exigences entre différents projets logiciels motive l'existence d'outils différents prenant en compte des préoccupations similaires, mais chacun avec des spécificités permettant de répondre au mieux aux exigences des différents projets. Ceci permet de comprendre le nombre d'outils existants et l'utopie de viser l'émergence d'un unique outil. Ainsi, certaines équipes de développement font le choix de l'IDE Visual Studio pour développer des applications Windows afin de profiter de toutes les facilités proposées par Microsoft pour ce faire, alors que si l'application à développer a des contraintes d'indépendance vis-à-vis du système d'exploitation sur lequel elle sera déployée, une équipe de développement pourra préférer l'IDE Eclipse pour faire du développement en Java.

L'utilisation de tous ces outils est généralement bénéfique au bon fonctionnement d'un projet mais elle engendre aussi un ensemble de tâches d'ingénierie logicielle qui sont réalisées manuellement par les équipes de développement. Utiliser ces différents outils peut impliquer de devoir s'en procurer une version, de devoir les installer, les configurer, ou encore les mettre à jour. Par exemple, un développeur utilisant un IDE peut être amené à installer les librairies et les *plug-ins* nécessaires pour un projet, ainsi qu'à configurer ces derniers. Ces tâches sont de plus récurrentes dans le sens où elles peuvent avoir lieu plusieurs fois pendant un même projet mais aussi qu'elles se répètent d'un projet à l'autre, toujours dans un contexte différent. Par exemple, dans un projet, chaque développeur doit installer un IDE conforme aux exigences de ce projet. Nous appelons dans cette thèse ces tâches répétitives et réalisées manuellement des *Tâches Manuelles Récurrentes* (TMR). Ces TMR, bien qu'étant toujours similaires, doivent prendre en compte le contexte particulier dans lequel elles sont réalisées. Elles se révèlent être coûteuses en temps et sources d'erreurs pour les équipes de développement [RCB+11].

1.2 Le défi : réutiliser des automatisations de tâches manuelles récurrentes

L'automatisation des TMR liées à l'utilisation des outils de développement logiciel doit permettre de gagner en productivité. Cela doit aussi permettre d'éviter les erreurs humaines inhérentes à toute activité répétitive. Cependant, l'intérêt de la mise en place d'une telle automatisation est limité si les automatisations de TMR ne sont pas réutilisées, au sein d'un même projet mais également d'un projet à l'autre. Or, la réutilisation d'automatisations de TMR est un exercice difficile pour les trois raisons suivantes :

1. Bien qu'une automatisation de TMR puisse être utile pour différents projets, toutes les automatisations de TMR ne sont pas forcément utiles pour chaque projet. La difficulté est donc de déterminer quelles automatisations de TMR réutiliser pour un projet donné.
2. Une automatisation de TMR pouvant avoir des dépendances avec d'autres tâches d'un projet, une difficulté supplémentaire est de déterminer à quels moments d'un projet utiliser une automatisation de TMR.

3. La problématique du niveau de généralisation est aussi un problème complexe. Comment déterminer qu'une automatisation de TMR utile pour des projets différents, ou à des moments différents d'un même projet, soit bien réutilisable à travers ces différents cas d'utilisation ?

1.3 Contributions de la thèse

Afin de répondre aux défis précédemment énoncés, la contribution de cette thèse est de piloter l'automatisation des TMR liées à l'utilisation des outils de développement logiciel par les processus de développement logiciel. Un processus de développement logiciel correspond à l'ensemble des activités d'ingénierie logicielle requises pour transformer les exigences d'un utilisateur en logiciel [Hum88].

La mise en œuvre de cette contribution s'appuie sur la définition d'une famille de processus de développement logiciel et sur la réutilisation des processus de cette famille en fonction des exigences des projets. Des formalismes différents pouvant être utilisés pour définir les processus de développement logiciel en fonction des exigences des utilisateurs [Zam01], nous proposons une approche (première sous-contribution) permettant de réutiliser les processus indépendamment du langage utilisé pour les définir. Cette approche repose sur l'ingénierie des lignes de produits [PBL05] (où ici les produits sont des processus de développement logiciel) afin de gérer la variabilité dans les processus.

La mise en œuvre de la contribution consiste également à lier des automatisations de TMR aux unités de travail d'une famille de processus qu'elles automatisent. Une même automatisation de TMR peut être liée à des unités de travail différentes, appartenant ou non à un même processus. Nous proposons donc une méthodologie (deuxième sous-contribution) fournissant un support à la création d'automatisations de TMR qui soient réutilisables pour toutes les unités de travail auxquelles elles sont liées. L'idée consiste à utiliser le lien entre les automatisations de TMR et la famille de processus pour identifier ces différentes unités de travail. Cette information est ensuite utilisée afin d'implémenter des automatisations réutilisables pour toutes ces unités de travail.

Lors de la réalisation d'un processus d'une famille, le lien entre les processus et les automatisations de TMR est utilisé afin de savoir quelles automatisations exécuter pour un projet donné et quand.

Pour réaliser cette contribution, nous nous appuyons sur l'IDM (Ingénierie Dirigée par les Modèles) [JCV12] qui offre les outils et méthodes nécessaires pour obtenir une représentation holistique et abstraite des différentes préoccupations (exigences, variabilité et processus) et raisonner de manière uniforme au travers de celles-ci.

Afin de valider notre contribution, nous avons développé un outillage supportant sa mise en œuvre. La validation de la contribution a été démontrée sur une famille de processus industriels de développement web Java issue de la société Sodifrance, ainsi que sur une famille de processus consistant à définir et outiller un langage de modélisation.

1.4 Contexte de réalisation de la thèse

Cette thèse s'est déroulée dans le cadre d'un contrat Cifre avec la société Sodifrance. Cela nous a permis d'appréhender concrètement la difficulté majeure à laquelle sont confrontées les ESN (Entreprises de Services du Numérique) du secteur du tertiaire. En effet, afin de faire face à la concurrence, ces sociétés sont dans une situation paradoxale où elles doivent réduire leur coûts et temps de développement, tout en garantissant la qualité des produits livrés au client. C'est d'ailleurs cette difficulté qui justifie ces travaux de thèse. Ceux-ci s'inscrivent en effet dans une démarche d'augmentation de la productivité des équipes de développement de Sodifrance. Bénéficier d'une collaboration industrielle a également fait ressortir les exigences principales auxquelles des travaux de recherche doivent répondre pour être utilisés dans l'industrie : être pertinents, outillés et validés. Des travaux de recherche n'ont en effet d'intérêt pour une entreprise que s'ils apportent une solution à des problèmes rencontrés par cette entreprise. Le coût de production élevé de l'outillage support à des travaux de recherche peut de plus empêcher l'adoption d'une approche. D'autre part, les contraintes financières auxquelles sont souvent confrontées les entreprises peuvent être un frein à l'adoption d'une approche qui n'a pas fait ses preuves. Ce sont ces exigences qui ont permis d'orienter la réalisation de ces travaux de thèse et de déterminer les améliorations à leur apporter. Enfin, la collaboration avec Sodifrance nous a permis d'avoir accès à des cas concrets de projets de développement, à partir desquels nous avons pu concevoir, réaliser et appliquer les contributions de cette thèse.

1.5 Plan de la thèse

Cette thèse s'organise en trois parties principales, qui sont le contexte scientifique (partie I), les contributions (partie II) et la validation des travaux de thèse (partie III).

Le contexte scientifique (partie I) s'organise en deux chapitres. Le chapitre 2 introduit l'environnement scientifique sur lequel s'appuient ces travaux de thèse. Plus précisément, il introduit l'IDM, les processus de développement logiciel, ainsi que l'ingénierie des lignes de produits logiciels. Il introduit également SPEM et CVL, deux langages de l'OMG permettant respectivement de définir des processus de développement logiciel et de mettre en œuvre l'ingénierie des lignes de produits. Le chapitre 3, traite de l'état de l'art sur la réutilisation d'automatisations de TMR. Cet état de l'art porte plus particulièrement sur les approches permettant de savoir quand réutiliser des automatisations de TMR et permettant de développer des composants logiciels réutilisables.

La contribution de la thèse, le pilotage de l'automatisation des TMR par les processus de développement logiciel, est présentée dans la partie II. Les sous-contributions de la thèse sont détaillées dans deux chapitres. Le chapitre 4 présente la première sous-contribution, à savoir notre approche permettant de réutiliser les processus de développement logiciel en fonction des exigences des projets et indépendamment du langage utilisé pour définir ces processus. Le chapitre 5 détaille la deuxième sous-

contribution, une méthodologie fournissant un support à la création d'automatisations de TMR réutilisables dans tous leurs cas d'utilisation.

La validation des travaux de thèse (partie III) comprend deux chapitres. Dans le chapitre 6, nous présentons l'outillage support aux contributions de cette thèse. Dans le chapitre 7, nous appliquons cet outillage à une famille de processus consistant à définir et outiller un langage de modélisation ainsi qu'à une famille de processus de développement web Java issue de Sodifrance.

Enfin, nous concluons et présentons nos perspectives de travail dans le chapitre 8.

Première partie

Contexte scientifique

Dans cette partie, nous commençons par présenter dans le chapitre 2 les différents champs de recherche sur lesquels nous nous appuyons dans cette thèse. Nous présentons ensuite dans le chapitre 3 l'état de l'art de cette thèse.

Chapitre 2

Background

Nous présentons dans ce chapitre les différents champs de recherche sur lesquels nous nous appuyons dans cette thèse. Nous commençons par présenter, dans la section 2.1, l'IDM (Ingénierie Dirigée par les Modèles), que nous utilisons pour mettre en œuvre les contributions de cette thèse. Nous présentons ensuite, dans la section 2.2, les processus de développement logiciel, ainsi que l'ingénierie des lignes de produits logiciels dans la section 2.3. En plus d'introduire, dans les sections 2.2 et 2.3, les concepts relatifs aux processus de développement logiciel et à l'ingénierie des lignes de produits, nous montrons comment l'IDM s'applique à ces domaines. Enfin, nous faisons une synthèse de ce chapitre dans la section 2.4.

2.1 L'ingénierie dirigée par les modèles (IDM)

La séparation des préoccupations [Dij82] est un moyen de gérer la complexité des logiciels. En effet, il est plus simple et efficace pour l'esprit humain de résoudre un problème complexe en le décomposant en sous-problèmes de complexité moindre plutôt que de considérer le problème complexe dans son ensemble.

L'IDM [JCV12] est un moyen de mettre en œuvre la séparation des préoccupations, en s'appuyant sur le principe de l'abstraction. En effet, le principe général de l'IDM est d'utiliser une représentation simplifiée d'un aspect du monde réel (c'est-à-dire un *modèle*) dans un objectif donné, en s'abstrayant des détails qui n'ont pas de rapport avec cet aspect. Dans le domaine de l'ingénierie logicielle, chaque aspect d'un système logiciel peut donc être représenté par un modèle. Ce modèle est lui-même exprimé avec un langage de modélisation généraliste tel qu'UML [OMG11b, OMG11c] ou avec un langage de modélisation dédié à un domaine métier particulier. L'intérêt de ces langages est qu'ils soient plus simples à appréhender pour des humains que les langages de programmation classiques (Java, C#, ...) ou que le langage machine. Un modèle peut alors être vu comme un support plus adapté pour des humains pour exprimer un aspect particulier d'un logiciel ainsi que pour réfléchir et communiquer sur cet aspect.

Une problématique clé de l'IDM est également de rendre les modèles opération-

nels. Cependant, à cette fin, ceux-ci doivent être interprétables par des machines. Le langage dans lequel ils sont exprimés doit donc être clairement défini. Comme tout langage informatique, un langage de modélisation est caractérisé par sa syntaxe abstraite, sa syntaxe concrète et sa sémantique [JCV12]. La syntaxe abstraite est la syntaxe manipulée par l'ordinateur. Dans le contexte des langages de modélisation, elle sert généralement de base à la définition de la syntaxe concrète et de la sémantique. La syntaxe concrète est la syntaxe manipulée par l'utilisateur du langage à l'aide d'un éditeur. Elle comprend des éléments syntaxiques dont le but est de faciliter l'utilisation d'un langage pour des humains. Elle associe des représentations pour chacun des éléments de la syntaxe abstraite. Il est tout à fait possible de définir des syntaxes concrètes différentes pour une même syntaxe abstraite, en fonction des besoins. La sémantique d'un langage de modélisation est définie en associant un sens à chacun des éléments de la syntaxe abstraite. Il est également possible de définir des sémantiques différentes pour une même syntaxe abstraite, en fonction des besoins (ex : vérification, simulation, compilation, etc.).

Dans le domaine de l'IDM, la syntaxe abstraite peut prendre la forme d'un modèle, appelé *métamodèle*. On dit d'un modèle m qu'il est conforme à un métamodèle mm si chaque élément de m est une instance d'un élément de mm [Guy13]. Cependant, face à la prolifération des métamodèles, le besoin est apparu de les unifier avec un langage de description commun, afin de les rendre compatibles entre eux et de pouvoir définir des outils capables de les manipuler de manière générique. L'OMG a donc proposé un langage de description des métamodèles, qui prend lui aussi la forme d'un modèle : le *métamétamodèle* MOF (*Meta-Object Facility*) [OMG06]. La particularité du métamétamodèle est qu'il se définit lui-même, limitant ainsi le nombre de niveaux de modélisation. Quand le métamétamodèle MOF ne permet pas d'exprimer toutes les propriétés d'un métamodèle, il est possible d'utiliser un langage de contrainte, tel qu'OCL (*Object Constraint Language*) [OMG10], afin d'exprimer ces propriétés.

Dans la pratique, l'utilisation de modèles de manière opérationnelle peut prendre plusieurs formes. Nous ne présentons ici que celles qui sont pertinentes pour la compréhension de cette thèse. Ainsi, les modèles peuvent être utilisés directement comme le système logiciel final à produire, en les exécutant tout comme il est possible d'exécuter n'importe quel programme informatique. Deux méthodes existent pour exécuter des modèles : l'interprétation et la compilation. L'interprétation consiste à analyser chaque élément d'un modèle et à l'exécuter en fonction de sa sémantique. La compilation consiste à transformer un modèle conforme à un langage en un modèle ou du texte conforme à un autre langage, et ensuite à exécuter le modèle ou le texte obtenu après transformation.

Il existe à ce jour de nombreux outils support à l'IDM. Dans cette thèse, nous utilisons EMF (*Eclipse Modeling Framework*) [SBPM09], un *framework* de l'environnement de développement Eclipse [1], afin de définir des métamodèles, de générer des éditeurs arborescents permettant de créer des modèles conformes à ces métamodèles, ainsi que de manipuler ces modèles et métamodèles en Java. EMF s'appuie sur le langage de

1. http://www.eclipse.org

métamodélisation Ecore [BSE03], qui est similaire au MOF. Nous faisons également référence dans cette thèse à l'utilisation d'UML afin de définir des langages de modélisation dédiés à un domaine métier particulier. Le mécanisme de *profil* UML permet en effet d'étendre UML de façon générique. Plus précisément, il permet d'ajouter de nouveaux concepts propre à un domaine métier particulier, de définir leurs propriétés, de spécifier des contraintes sur ces concepts et également de leur associer une représentation graphique. Un profil est définit principalement à l'aide de *stéréotypes*, où un stéréotype définit un nouveau concept, mais toujours sous la forme d'une extension d'un élément UML. Un stéréotype peut avoir des propriétés ainsi qu'une représentation graphique. Des contraintes (telles que des contraintes OCL) peuvent également être définies sur ce stéréotype.

2.2 Les processus de développement logiciel

2.2.1 Introduction aux processus de développement logiciel

Un processus de développement logiciel correspond à l'ensemble des activités d'ingénierie logicielle requises pour transformer les exigences d'un utilisateur en logiciel [Hum88]. La maîtrise de ces processus est un enjeu crucial pour les entreprises qui développent des logiciels. Pour preuve, le CMM (*Capability Maturity Model*) [Ins95] est une méthodologie qui se base sur l'amélioration des processus de développement logiciel afin d'améliorer la qualité des logiciels produits, tout en réduisant les coûts et les temps de production [HZG+97, Dio93]. La maîtrise des processus de développement logiciel passe, entre autres, par l'utilisation de formalisations de processus afin de supporter leur exécution [Ost87, Ins95]. L'exécution d'un processus de développement logiciel consiste en la réalisation des différentes unités de travail définies par ce processus, dans l'ordre dans lequel elles sont définies [Ben07].

Des méthodes [2] de développement logiciel, telles que les méthodes de développement en cascade [Roy87] ou en spirale [Boe88], ont dans un premier temps été proposées afin de définir les processus de développement logiciel. Ces méthodes donnent une vue générale des activités à accomplir pour mener à bien un projet de développement logiciel. Mais leur principale limitation est qu'elles ne fournissent pas assez de détails pour guider la réalisation concrète de ces activités [CKO92]. La notion de modèle de processus est alors apparue afin de faire face à cette limitation, où un modèle de processus capture de manière plus détaillée les informations nécessaires pour guider la réalisation d'un projet. Ces modèles de processus sont exprimés selon un langage de modélisation de processus de développement logiciel. Il existe à ce jour plusieurs langages de modélisation de processus de développement logiciel, comme SPEM [OMG08], xSPEM [BCCG07], Little-JIL [Wis06], UML4SPM [BGB06], ou encore SEMDM [ISO07]. Nous présentons plus en détails un de ces langages, SPEM, en section 2.2.2.

2. Une méthode définit comment mettre en œuvre des unités de travail et des artéfacts qui servent à la définition de processus de développement logiciel [HIMT96].

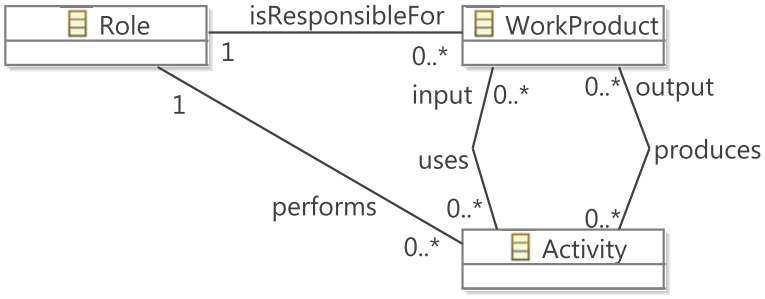


FIGURE 2.1 – Modèle conceptuel de SPEM

Un modèle de processus peut ensuite servir à guider l'exécution d'un processus. En effet, le modèle de processus peut être utilisé comme support de communication entre les différents membres d'une équipe [Ost87, Ins95]. Un modèle de processus peut aussi être utilisé comme un programme qu'il est possible d'exécuter [Ost87], afin d'automatiser les étapes de l'exécution du processus qui peuvent l'être [RGC04]. Des outils supportant l'exécution de langages de modélisation de processus ont ainsi vu le jour, comme Wilos [3] pour SPEM, Juliette [CLM+99, CLS+00] pour Little-JIL, ainsi qu'un compilateur et un interpréteur [Ben07] pour UML4SPM.

2.2.2 Modélisation des processus de développement logiciel avec SPEM 2.0

SPEM 2.0 (*Software & Systems Process Engineering Metamodel*) [OMG08] est le standard de l'OMG dédié à la modélisation des processus logiciels et systèmes. Il vise à offrir un cadre conceptuel pour modéliser, échanger, documenter, gérer et présenter les processus et méthodes.

Les concepts de SPEM 2.0 sont décrits par un métamodèle qui repose sur l'idée qu'un processus de développement est une collaboration entre des entités actives et abstraites, les *rôles*, qui représentent un ensemble de compétences et qui effectuent des opérations, les *activités*, sur des entités concrètes et réelles, les *produits*. Les différents rôles agissent les uns par rapport aux autres ou collaborent en échangeant des produits et en déclenchant l'exécution de certaines activités. Ce modèle conceptuel est synthétisé sur la figure 2.1.

La particularité de SPEM 2.0 est qu'il sépare les aspects contenus et données relatifs aux méthodologies de développement, de leurs possibles instanciations dans un projet particulier. Ainsi, pour pleinement exploiter ce *framework*, la première étape devrait être de définir toutes les phases, activités, produits, rôles, guides, outils, etc., qui peuvent composer une méthode pour ensuite, dans une deuxième étape, choisir en fonction du contexte et du projet, le contenu de la méthode appropriée pour l'utiliser dans la définition du processus.

SPEM 2.0 est défini sous la forme d'un métamodèle MOF [OMG06] qui s'appuie sur les spécifications UML 2.0 Infrastructure [OMG11b] et UML 2.0 Diagram Defini-

3. www.wilos.ups-tlse.fr

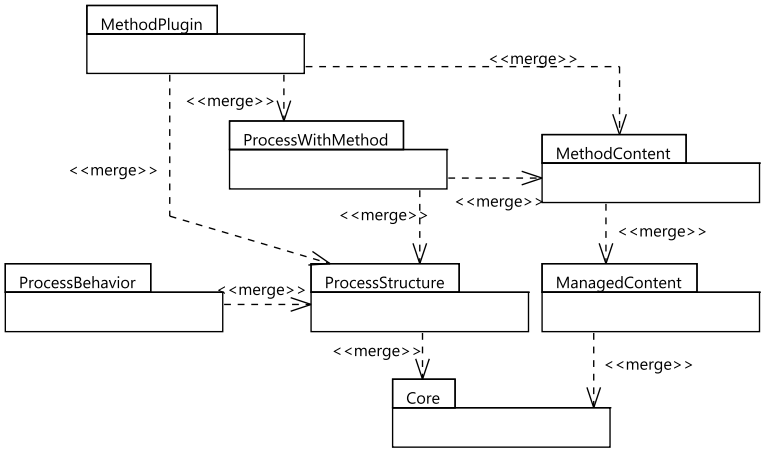


FIGURE 2.2 – Structure du métamodèle de SPEM 2.0 [OMG08]

tion [OMG12]. De UML 2.0 Infrastructure, il réutilise les concepts de base tels que `Classifier` ou `Package`. Aucun concept de UML 2.0 Superstructure [OMG11c] n'est réutilisé. Le standard est défini également sous la forme d'un profil UML où chaque élément du métamodèle de SPEM 2.0 est défini comme un stéréotype de UML 2.0 Superstructure.

Le métamodèle de SPEM 2.0 est composé de sept paquetages liés avec le mécanisme «merge» [OMG11b], chaque paquetage traitant d'un aspect particulier (cf. figure 2.2). Le paquetage `Core` introduit les classes et les abstractions qui définissent les fondations pour tous les autres paquetages. La structure de décomposition de ce paquetage est la classe `WorkDefinition`, qui généralise toutes les activités de travail de SPEM 2.0. Le paquetage `ProcessStructure` définit les éléments permettant de représenter les modèles de processus. Cependant, la possibilité de documenter textuellement ces éléments (c.-à-d. ajouter les propriétés décrivant chaque élément) n'est pas fournie dans ce paquetage mais dans `ManagedContent`, qui définit les concepts pour gérer les descriptions textuelles des éléments de processus. Des exemples de ces concepts sont les classes `ContentDescription` et `Guidance`. Le paquetage `MethodContent` définit les concepts de base pour spécifier le contenu de méthodes basiques. Le paquetage `ProcessWithMethod` regroupe l'ensemble des éléments requis pour intégrer les processus définis avec les concepts du paquetage `ProcessStructure`, selon les contenus définis avec les concepts du paquetage `MethodContent`. Le paquetage `MethodPlugin` offre les mécanismes pour gérer et réutiliser des bibliothèques de contenus de méthode et processus. Ceci est assuré grâce aux concepts `MethodLibrary` et `MethodPlugin`. Une instance de `MethodLibrary` est un conteneur pour tout élément SPEM 2.0. C'est donc l'élément racine d'un modèle SPEM 2.0. Une instance de `MethodPlugin` est plus particulièrement un conteneur pour des contenus de méthodes et des processus. Enfin, le paquetage `ProcessBehavior` offre le moyen de lier un élément de procédé SPEM 2.0 avec un comportement externe comme des diagrammes d'activités UML 2.0 ou des modèles BPMN (*Business Process Modeling Notation*) [OMG11a].

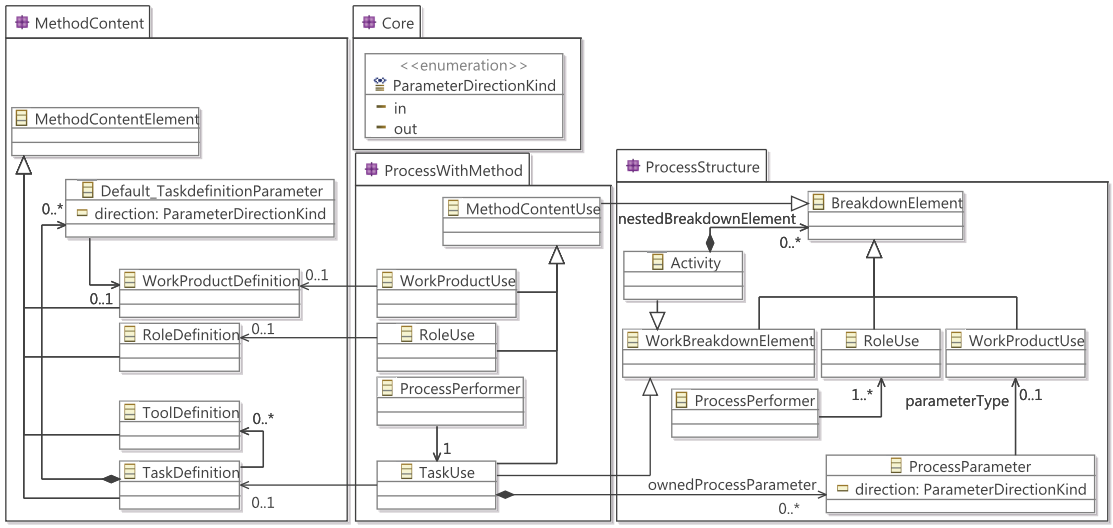


FIGURE 2.3 – Extraits de certains paquetages du métamodèle de SPEM 2.0

La figure 2.3 détaille le contenu des paquetages `Core`, `MethodContent`, `ProcessStructure` et `ProcessWithMethod` qui est utile pour la compréhension de la suite de cette thèse.

Le paquetage `MethodContent` contient des éléments de contenu de méthode (`MethodContentElement`), tels que des définitions de rôles (`RoleDefinition`), qui réalisent des définitions de tâches (`TaskDefinition`) en utilisant des définitions d'outils (`ToolDefinition`). Une définition de tâche a des définitions de produits de travail (`WorkProductDefinition`) en tant qu'entrées et sorties.

Le paquetage `ProcessStructure` définit des éléments de processus appartenant à une structure de décomposition (`BreakdownElement`). Parmi ces éléments de processus sont définis des produits de travail (`WorkProductUse`), des rôles (`RoleUse`), et également des éléments représentant un travail (`WorkBreakdownElement`). Parmi les éléments représentant un travail, une activité (`Activity`) est un conteneur pour d'autres éléments de processus appartenant à une structure de décomposition.

Le paquetage `ProcessWithMethod` spécifie qu'un produit de travail se base sur une définition de produit de travail et qu'un rôle se base sur une définition de rôle. Il spécifie également un nouveau type d'élément de processus représentant un travail, la tâche (`TaskUse`), qui se base sur une définition de tâche. Une tâche est réalisée par un rôle, et c'est la classe `ProcessPerformer` qui fait le lien entre les deux. Une tâche a également des produits de travail en entrée et en sortie, et c'est un paramètre (classe `ProcessParameter` du paquetage `ProcessStructure`) qui permet de faire le lien entre une tâche et ses entrées et sorties.

Le paquetage `Core` contient entre autres une classe `ParameterDirectionKind`, qui permet de définir si un produit de travail est une entrée (`in`) ou une sortie (`out`) d'un travail.

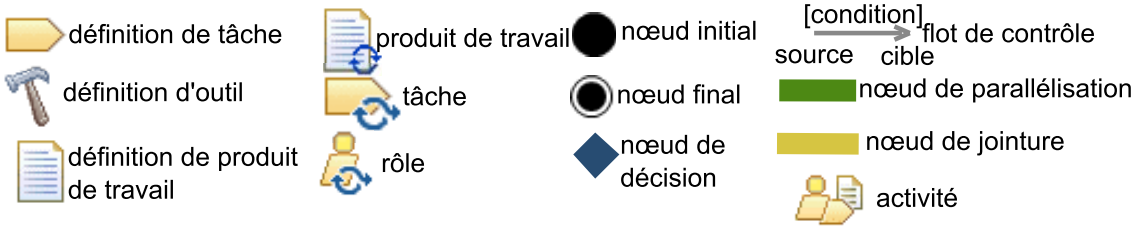


FIGURE 2.4 – Syntaxe concrète de SPEM 2.0 et des diagrammes d'activités

SPEM 2.0 définit également un plug-in de base, sous la forme d'une instance de `MethodPlugin`. Ce plug-in définit un type particulier d'activité, le processus (`DeliveryProcess`), qui correspond à l'approche complète pour réaliser un type de projet particulier. Un processus définit donc la séquence d'activités à réaliser pour mener à bien un projet de développement logiciel, tandis qu'une activité définit la séquence de tâches à effectuer pour réaliser cette activité.

Étant donné qu'en SPEM 2.0 les tâches et les activités sont des actions (`Action`) UML 2.0 stéréotypées, il est possible d'utiliser les diagrammes d'activités UML 2.0 pour modéliser une séquence d'activités ou de tâches. En UML 2.0, une action, un flot de contrôle (`ControlFlow`), un nœud initial (`InitialNode`), un nœud final (`ActivityFinalNode`), un nœud de jointure (`JoinNode`), un nœud de parallélisation (`ForkNode`) et un nœud de décision (`DecisionNode`) sont des sous-types de nœud d'activité (`ActivityNode`). Un flot de contrôle représente l'enchaînement de deux nœuds d'activité, où un nœud d'activité source est suivi par un nœud d'activité cible. La condition d'un flot de contrôle, quand elle existe, spécifie à quelle condition ce flot de contrôle est navigable. Les nœuds initiaux et finaux spécifient respectivement le début et la fin d'un enchaînement d'activités ou de tâches. Un nœud de parallélisation divise un flot de contrôle en plusieurs flots de contrôle navigables simultanément. Un nœud de jointure fusionne plusieurs flots de contrôle en un seul. Un nœud de décision divise un flot de contrôle en plusieurs flots de contrôle alternatifs.

La figure 2.4 illustre la syntaxe concrète que nous utilisons dans cette thèse pour représenter des processus SPEM 2.0 ainsi que des diagrammes d'activités.

Pour terminer, nous illustrons l'utilisation de SPEM 2.0 en modélisant un exemple de processus simplifié de développement d'une application Java. Ce processus (cf. figure 2.5) commence avec l'activité *spécifier*, durant laquelle un *expert fonctionnel* produit un document définissant les spécifications techniques et fonctionnelles de l'application à développer, en fonction des exigences d'un projet. L'activité *spécifier* est suivie par l'activité *développer*, pendant laquelle un *développeur* code manuellement l'application. Lors de l'activité *tester* suivante, un testeur vérifie que l'application en cours de développement est bien conforme aux spécifications. Si cette activité révèle des erreurs, celles-ci doivent être corrigées, ce qui se traduit dans le processus par un retour à l'activité de développement. S'il n'y a pas d'erreur, le processus se termine par l'activité *mettre en production*, qui consiste à déployer l'application sur l'environnement du client. Dans le cas de cet exemple, l'activité de développement consiste en la création et la

modification de projets Eclipse, comme illustré par la figure 2.6. Les tâches *créer premier projet Eclipse*, *créer projet Eclipse*, *modifier projets Eclipse* ainsi que le produit de travail *projet(s) Eclipse* font respectivement référence aux éléments de contenu de méthode de même nom de la figure 2.7.

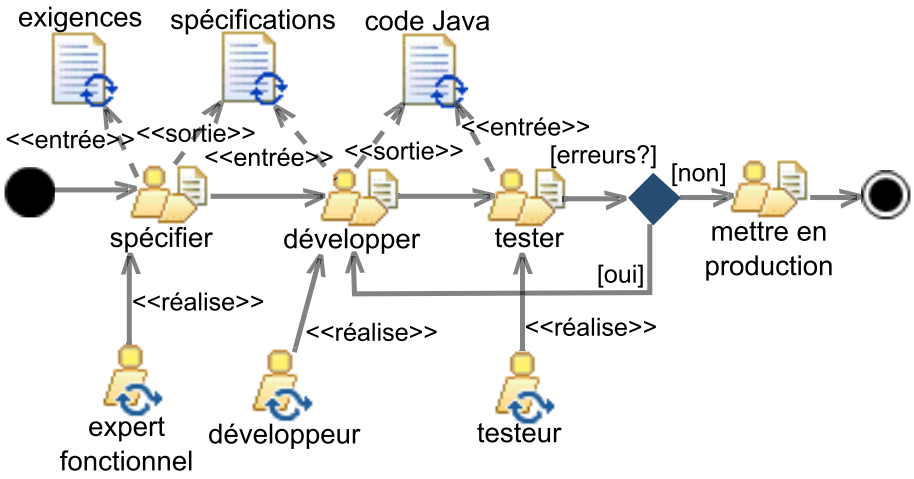


FIGURE 2.5 – Un processus simplifié de développement Java

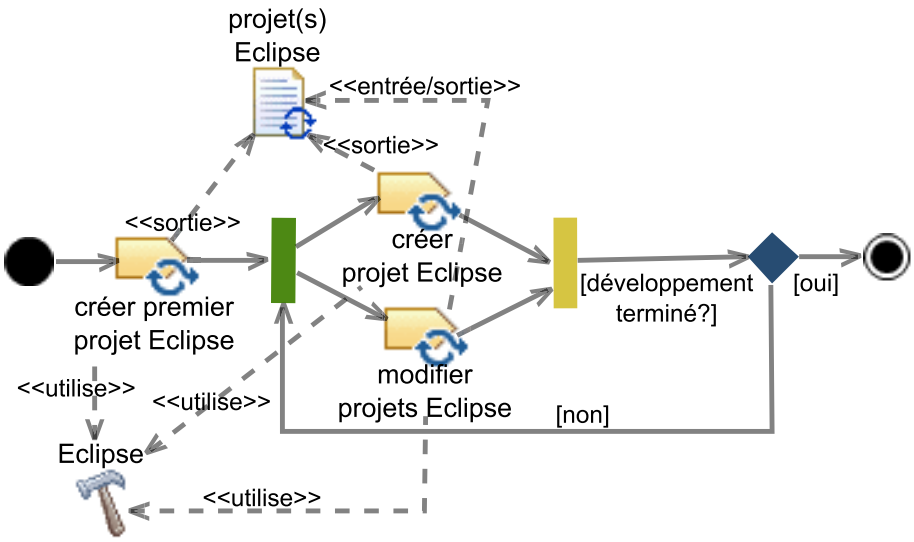


FIGURE 2.6 – Détail de l'activité de développement du processus de la figure 2.5

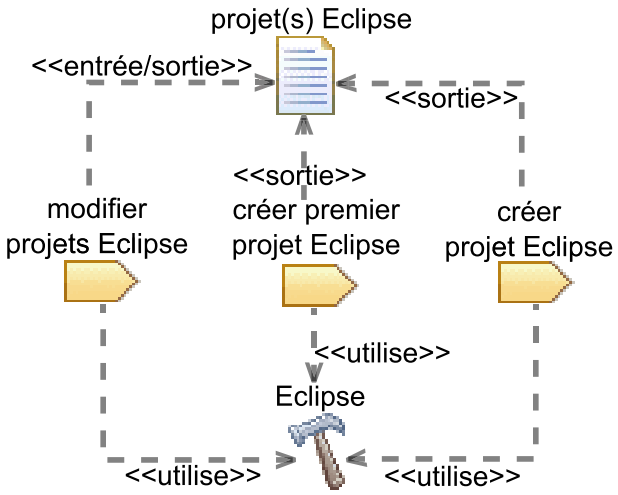


FIGURE 2.7 – Éléments de contenu de méthode du processus simplifié de développement Java de la figure 2.5

2.3 L'ingénierie des lignes de produits logiciels

2.3.1 Introduction

Une *ligne de produits logiciels*, ou *famille de produits logiciels*, est un ensemble de produits logiciels partageant des caractéristiques communes et développés à partir d'un ensemble commun d'artéfacts [CN01]. L'ingénierie des lignes de produits logiciels [WL99, CN01, PBL05] permet la réutilisation d'artéfacts logiciels pour créer un nouveau produit logiciel, plutôt que de développer ce produit à partir de rien [Kru92]. Elle s'appuie sur l'identification de la variabilité et des parties communes d'une ligne de produits afin de réutiliser ces parties communes. L'ingénierie des lignes de produits logiciels apporte des bénéfices en termes de réduction des coûts et des temps de développement ainsi qu'en termes d'amélioration de la qualité des produits logiciels.

Afin de permettre la réutilisation des artéfacts logiciels, l'ingénierie des lignes de produits logiciels s'appuie sur deux processus : l'*ingénierie de domaine* et l'*ingénierie d'application* [PBL05]. Le processus d'ingénierie de domaine consiste à identifier la variabilité et les parties communes d'une ligne de produits, ainsi qu'à développer les artéfacts communs aux produits de la ligne, en vue de leur réutilisation. Les artéfacts partagés par plusieurs produits ne sont définis qu'une seule fois, afin d'assurer leur réutilisation et d'en faciliter la maintenance [Kru06]. Le processus d'ingénierie d'application, aussi appelé processus de *dérivation* d'un produit logiciel, consiste à développer un produit logiciel spécifique à partir de la réutilisation des artéfacts développés pendant le processus d'ingénierie de domaine. Il s'appuie sur une phase de *résolution de la variabilité*, qui consiste à définir la configuration de produit souhaitée en spécifiant la valeur de chaque partie variable de la ligne de produits.

On distingue deux types de variabilité : variabilité *positive* et variabilité *néga-*

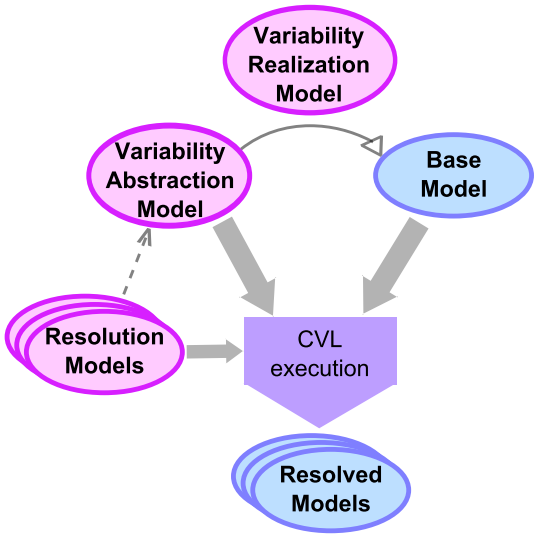


FIGURE 2.8 – Principe général de CVL, issu de la spécification de CVL [4]

tive [VG07]. La variabilité positive consiste, durant le processus d'ingénierie de domaine, à ne définir que les artéfacts communs à tous les produits d'une famille. Les artéfacts spécifiques à un produit sont quant à eux créés au moment de la dérivation de ce produit, en fonction de sa spécification. Au contraire, la variabilité négative consiste, durant le processus d'ingénierie de domaine, à définir tous les artéfacts nécessaires à la définition des produits d'une famille. Lors de la dérivation d'un produit, les artéfacts ne correspondant pas à ce produit ne sont donc pas sélectionnés.

Plusieurs approches s'appuient sur l'IDM afin de mettre en œuvre les concepts de l'ingénierie des lignes de produits logiciels [Jéz12]. L'IDM est ainsi utilisée pour définir des lignes produits et pour dériver des produits de ces lignes de produits. Nous présentons plus en détails une de ces approches, CVL [FHMP+11], dans la section suivante.

2.3.2 CVL

CVL (*Common Variability Language*) [4] [FHMP+11], un standard de l'OMG en cours de spécification, est un langage de modélisation qui permet de spécifier et de résoudre de la variabilité sur n'importe quel modèle dont le métamodèle est conforme au MOF.

Comme illustré en figure 2.8, CVL propose de définir un modèle de base (*base model*), qui est une instance de n'importe quel métamodèle conforme au MOF. Ce modèle de base contient les éléments de modèle nécessaires à la définition des différents produits d'une famille. CVL fournit également des constructions pour capturer la variabilité (VAM, *Variability Abstraction Model*) entre les différents produits d'une famille. Le VAM peut être comparé à un BFM (*Basic Feature Model*) [KCH+90], dans le sens où

4. La spécification complète de CVL peut être trouvée sur le wiki de CVL, à l'adresse http://www.omgwiki.org/variability/doku.php.

tout comme un BFM, il permet d'expliciter les différences et les similitudes entre les caractéristiques de produits logiciels d'une même famille. CVL fournit aussi des constructions pour définir la réalisation de cette variabilité sur le modèle de base (VRM, *Variability Realization Model*). Le VRM permet d'expliciter le lien entre le VAM et les éléments du modèle de base. Plus précisément, le VRM spécifie quels sont les éléments de modèle de base qui sont impactés par la variabilité définie dans le VAM et comment ces éléments de modèle sont impactés. En d'autres termes, le VRM spécifie comment dériver un produit du modèle de base, en fonction de la variabilité spécifiée dans le VAM. Enfin, CVL fournit des constructions permettant de résoudre la variabilité définie dans le VAM, afin de sélectionner une configuration du modèle de base (RM, *Resolution Model*).

Le VAM, le VRM et le RM contiennent les informations nécessaires à la dérivation, à partir du modèle de base, d'un modèle résolu (*resolved model*), c'est-à-dire sans variabilité. Le modèle résolu est alors une autre instance du métamodèle auquel le modèle de base est conforme.

Dans la suite, nous présentons les trois parties du métamodèle de CVL (abstraction de la variabilité, réalisation de la variabilité et résolution de la variabilité). Nous détaillons plus particulièrement les éléments du métamodèle de CVL qui seront utilisés pour illustrer la suite de cette thèse. Cependant, tous les éléments du métamodèle de CVL peuvent être utilisés pour mettre en œuvre ces travaux de thèse, en fonction des spécificités de chaque cas.

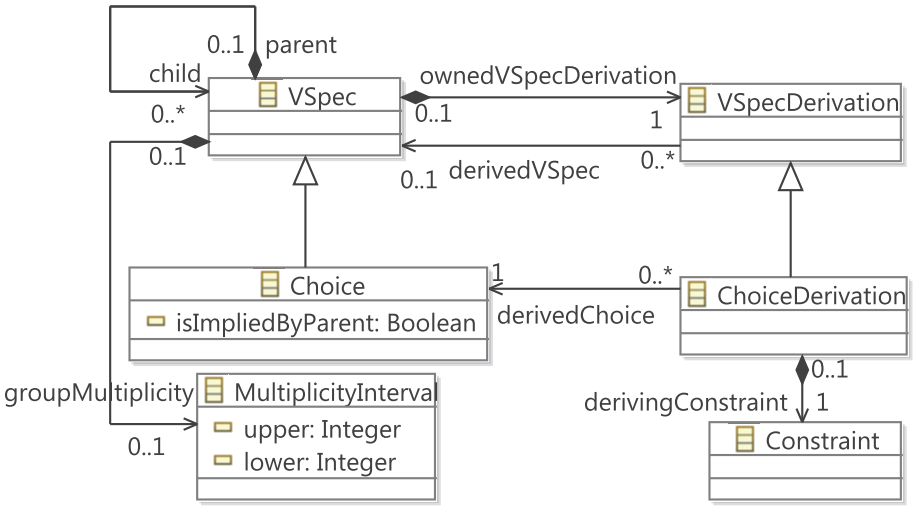


FIGURE 2.9 – Extrait de la partie abstraction de la variabilité du métamodèle de CVL

La figure 2.9 présente un extrait de la partie abstraction de la variabilité du métamodèle de CVL. Cette partie permet de définir de la variabilité à l'aide de *spécifications de variabilité* (`VSpec`). Une spécification de variabilité peut être un *choix* (`Choice`), c'est-à-dire une caractéristique qui appartiendra ou non au modèle résolu selon que ce choix sera résolu positivement ou négativement (c'est-à-dire selon que le choix sera sélectionné ou non). Une spécification de variabilité est également un conteneur pour

d'autres spécifications de variabilité, qui sont alors ses *enfants* (`child`). Ces enfants peuvent être résolus à vrai seulement si leur *parent* (`parent`) est résolu à vrai. Une spécification de variabilité a aussi une *multiplicité de groupe* (`groupMultiplicity`) qui définit les nombres minimum et maximum (attributs `lower` et `upper` de la classe `MultiplicityInterval`) d'enfants directs qui peuvent être résolus positivement. La multiplicité de groupe permet donc d'exprimer si des spécifications de variabilité ayant le même parent direct sont mutuellement exclusives ou non. Si un choix est *impliqué par son parent* (attribut `isImpliedByParent` à vrai), alors il doit forcément être sélectionné si son parent l'est. L'implication par le parent permet donc d'exprimer qu'une spécification de variabilité enfant est optionnelle ou obligatoire.

CVL fournit également le concept de *dérivation de spécification de variabilité* (`VSpecDerivation`), qui spécifie que la résolution d'une spécification de variabilité (identifiée par la référence `derivedVSpec`) est dérivée (c'est-à-dire calculée) de la résolution d'autres spécifications de variabilité. De ce fait, un RM ne définit pas de résolution pour une spécification de variabilité dont la résolution est dérivée. Un type particulier de dérivation de spécification de variabilité est la *dérivation de choix* (`ChoiceDerivation`), qui s'applique à un choix, identifié par la référence `derivedChoice`. Une dérivation de choix est associée à une *contrainte de dérivation* (`Constraint`) via la référence `derivingConstraint`. C'est cette contrainte de dérivation qui définit comment la résolution d'un choix est dérivée de la résolution d'autres spécifications de variabilité. Cette contrainte peut par exemple être exprimée à l'aide d'un langage de contrainte tel qu'OCL.

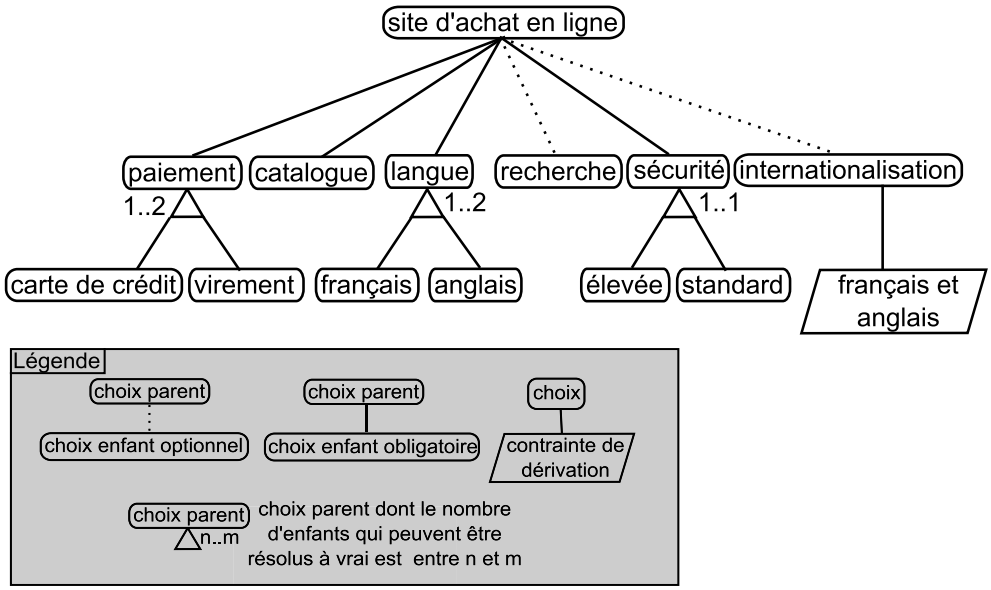


FIGURE 2.10 – Exemple VAM spécifiant la variabilité d'un site d'achat en ligne

À titre d'illustration (cf. figure 2.10), nous utilisons un VAM pour spécifier la variabilité d'un site d'achat en ligne. Les choix CVL permettent de spécifier les différentes

caractéristiques que peut avoir un tel site (*paiement*, *catalogue*, *langue*, *recherche*, *sécurité*, etc.). L'implication par le parent permet de spécifier les caractéristiques obligatoires (*paiement*, *catalogue*, *langue*, *sécurité*) et optionnelles (*recherche*, *internationalisation*). Une multiplicité de groupe dont la borne supérieure vaut 1 spécifie que des choix enfants sont mutuellement exclusifs (*élevé* et *standard*), tandis qu'une borne supérieure valant plus de 1 spécifie que plusieurs choix enfants peuvent appartenir à une même configuration de produit (par exemple *français* et *anglais*). La résolution du choix *internationalisation* est dérivée de la résolution des choix *français* et *anglais*. Plus précisément, un mécanisme d'internationalisation est utilisé pour développer un site d'achat en ligne si celui-ci doit être disponible dans les deux langues.

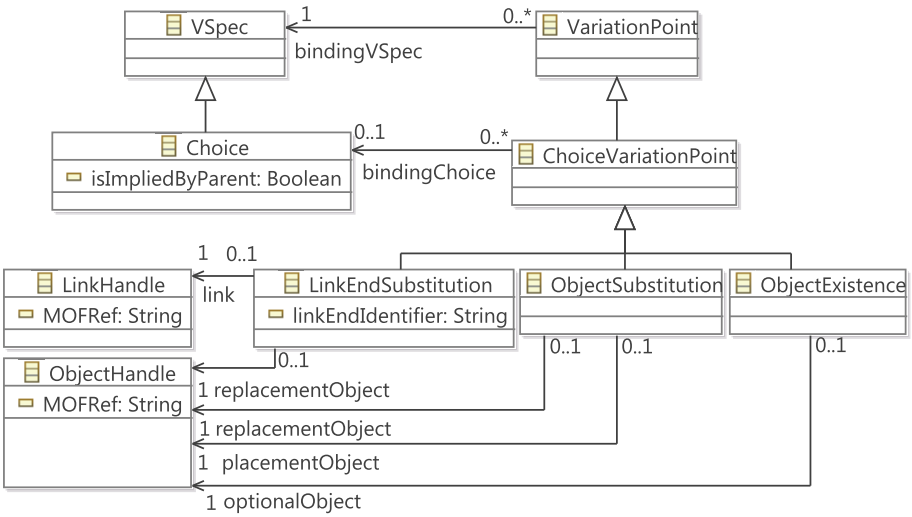


FIGURE 2.11 – Extrait de la partie réalisation de la variabilité du métamodèle de CVL

La figure 2.11 illustre un extrait de la partie réalisation de la variabilité du métamodèle de CVL. Le concept principal de cette partie réalisation de la variabilité est celui de *point de variation* (`VariationPoint`). Un point de variation est une opération à effectuer sur le modèle de base afin de dériver un modèle résolu. Un point de variation est effectué sur le modèle de base lorsque la spécification de variabilité qui lui est associée (`bindingVSpecs`) est résolue positivement.

CVL définit différents types de points de variation. Parmi eux, un *point de variation de choix* (`ChoiceVariationPoint`) est un point de variation associé à une spécification de variabilité de type choix. Un point de variation de choix s'applique donc lorsque le choix associé (`bindingChoice`) est sélectionné. Différents types de points de variation de choix existent. Parmi eux, une *substitution d'objet* (`ObjectSubstitution`) remplace un objet du modèle de base, l'*objet placement* (`placementObject`), par un autre objet du modèle de base, l'*objet remplaçant* (`replacementObject`), et supprime l'objet placement. Une *substitution de fin de lien* (`LinkEndSubstitution`) assigne une nouvelle valeur à un lien du modèle de base. Ce lien est identifié par le nom de la référence qu'il instancie (`linkEndIdentifier`) ainsi que par l'objet qui contient ce lien (`link`). La nouvelle

valeur à affecter au lien est un objet désigné par la référence `replacementObject` associée à la classe `LinkEndSubstitution`. Une *existence d'objet* (`ObjectExistence`) spécifie qu'un *objet optionnel* (`optionalObject`) du modèle de base existera toujours dans le modèle résolu. L'objet optionnel est supprimé du modèle de base (et donc du modèle résolu) si le choix associé au point de variation qui le contient n'est pas sélectionné. Ainsi, l'existence d'objet s'exécute en variabilité négative. Cela signifie qu'il n'est pas possible de créer des objets dans le modèle résolu. Si un objet appartient au modèle résolu, c'est forcément qu'il était défini dans le modèle de base. Il est uniquement possible de supprimer des objets du modèle de base pour qu'ils n'appartiennent pas à un modèle résolu. Au contraire, la variabilité positive signifie qu'il est possible de créer des objets au moment de la dérivation. Un *pointeur de lien* (`LinkHandle`) et un *pointeur d'objet* (`ObjectHandle`) permettent de référencer un objet du modèle de base via leur attribut `MOFRef`, qui correspond à l'URI de l'objet à référencer.

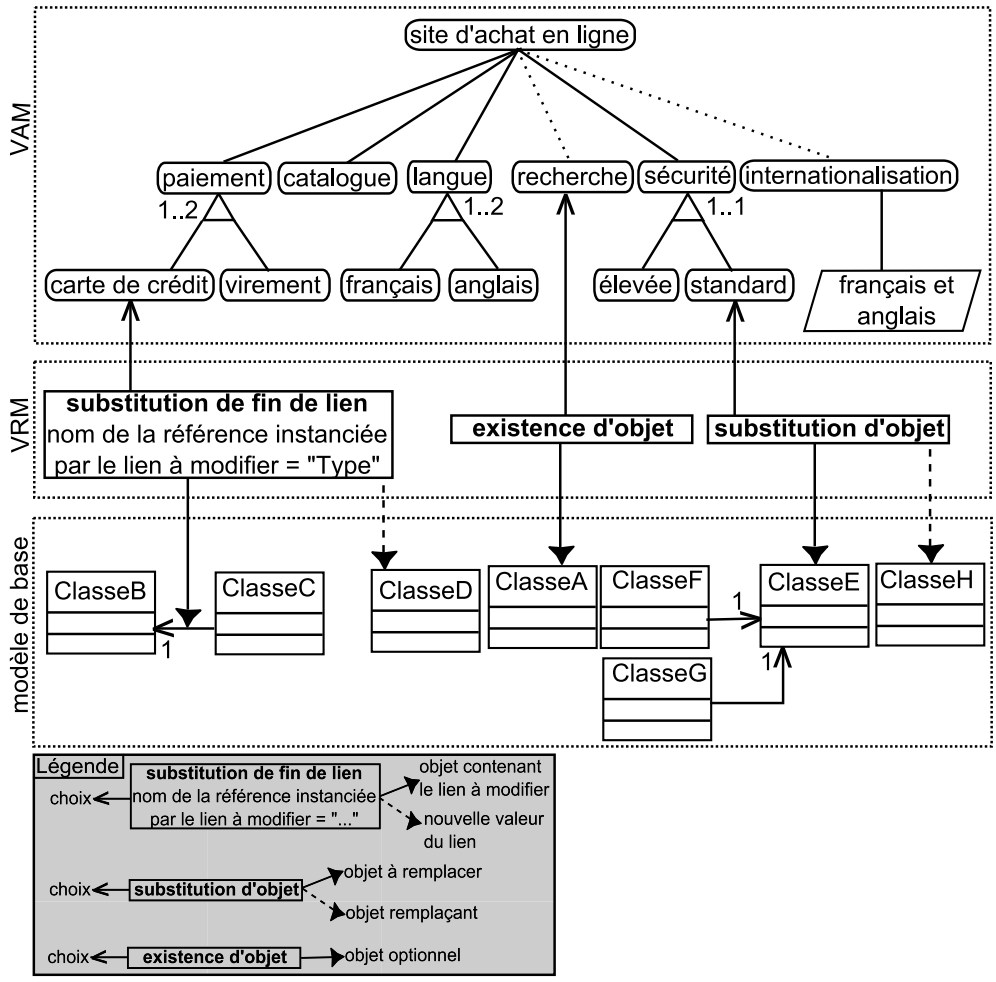


FIGURE 2.12 – VRM de l'exemple de site d'achat en ligne

Comme illustré par la figure 2.12, un VRM associé à l'exemple de site d'achat en

ligne spécifie que si le choix *carte de crédit* est sélectionné, alors, dans un modèle de base, la classe *ClasseC* doit référencer la classe *ClasseD* au lieu de la classe *ClasseB*. Si le choix *recherche* n'est pas sélectionné, alors la classe *ClasseA* du modèle de base appartiendra au modèle résolu. Enfin, si le choix *standard* est sélectionné, alors la classe *ClasseE* sera substituée par la classe *ClasseH* dans le modèle résolu.

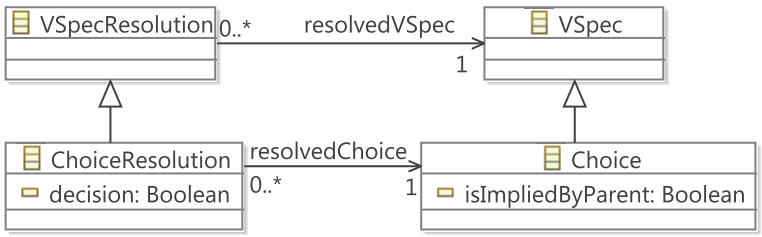


FIGURE 2.13 – Extrait de la partie résolution de la variabilité du métamodèle de CVL

La figure 2.13 présente un extrait de la partie résolution de la variabilité du métamodèle de CVL. Le concept central de cette partie est celui de *résolution de spécification de variabilité* (`VSpecResolution`), qui résout une spécification de variabilité identifiée par la référence `resolvedVSpec`. Un type particulier de résolution de spécification de variabilité est la *résolution de choix* (`ChoiceResolution`), qui résout un choix identifié par la référence `resolvedChoice`. L'attribut `decision` d'une résolution de choix permet d'indiquer la valeur de résolution d'un choix. Un choix est en effet sélectionné ou non selon que cet attribut vaut vrai ou faux.

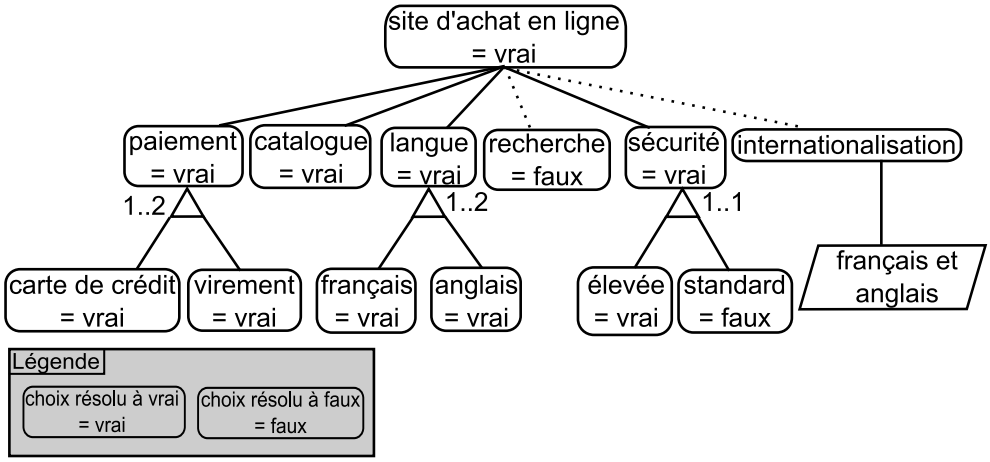


FIGURE 2.14 – Exemple de RM résolvant le VAM de la figure 2.10

Le RM présenté en figure 2.14 résout la variabilité du VAM de la figure 2.10. Tous les choix sont sélectionnés sauf les choix *recherche* et *standard*. En conséquence, les substitutions de fin de lien et d'objet du VRM de la figure 2.12 sont appliquées au modèle de base afin de dériver le modèle résolu (figure 2.15), tandis que l'existence d'objet n'est pas appliquée. Dans le modèle résolu, la classe *ClasseA* n'existe donc pas,

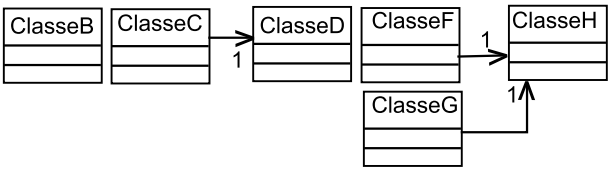


FIGURE 2.15 – Modèle résolu obtenu en fonction des modèles des figures 2.10, 2.12 et 2.14

la classe *ClasseC* référence la classe *ClasseD* au lieu de la classe *ClasseB* et la classe *ClasseE* a été substituée par la classe *ClasseH*.

CVL propose également d'autres concepts permettant de gérer la variabilité, dont nous ne nous servirons pas pour illustrer la suite de cette thèse, bien qu'ils puissent être utilisés pour mettre en œuvre ces travaux. Nous en donnons un aperçu dans la suite de cette section.

Il existe ainsi d'autres types de point de variation de choix. Parmi eux, l'*affectation d'attribut* permet d'assigner une nouvelle valeur à un attribut. L'*existence de lien* et l'*existence d'attribut*, de manière similaire à l'existence d'objet, permettent respectivement de spécifier l'existence d'un lien ou de la valeur d'un attribut. La figure 2.16 illustre un exemple d'utilisation de ces trois points de variation. Lorsque le choix *choix 1* est sélectionné, alors la classe *ClasseB* est renommée en *ClasseZ*, le nom de la classe *ClasseC* existe et cette dernière hérite de *ClasseZ* (cf. figure 2.17). À l'inverse, lorsque le choix *choix 1* n'est pas sélectionné, la classe *ClasseB* n'est pas renommée et le nom de la classe *ClasseC* est supprimé ainsi que la relation d'héritage (cf. figure 2.18). CVL permet également à ses utilisateurs de définir leurs propres points de variation, appelés *points de variation opaques*, en définissant des transformations d'un modèle vers un autre modèle.

CVL définit aussi des points de variation *paramétrables*, tels que la *substitution de fin de lien paramétrable*, l'*affectation d'attribut paramétrable* et la *substitution d'objet paramétrable*. Ceux-ci se comportent comme leurs homologues non paramétrables, à la différence qu'une valeur à assigner ou un objet remplaçant ne sont spécifiés qu'au moment de la résolution, dans le RM. Cela est utile lorsqu'un élément du modèle de base a des variantes qui ne sont pas connues au moment de l'édition du VRM. Un point de variation paramétrable est relié à un type de spécification particulier, la *variable*. La valeur assignée à une variable est affectée au moment de la résolution, en utilisant un type de résolution de spécification de variabilité particulier, l'*affectation de valeur à une variable*.

CVL propose également un point de variation dit *répétable*, la *substitution de fragment*. Exécutée une seule fois, la substitution de fragment remplace un fragment du modèle de base par un autre. Lorsqu'elle est répétée, une substitution de fragment crée une nouvelle instance du fragment remplaçant à chaque répétition. CVL définit un type de spécification de variabilité et un type de résolution de spécification de variabilité spécifiques au point de variation répétable, respectivement le *classificateur de variabilité* et l'*instance de variabilité*. Le classificateur de variabilité permet de spécifier de

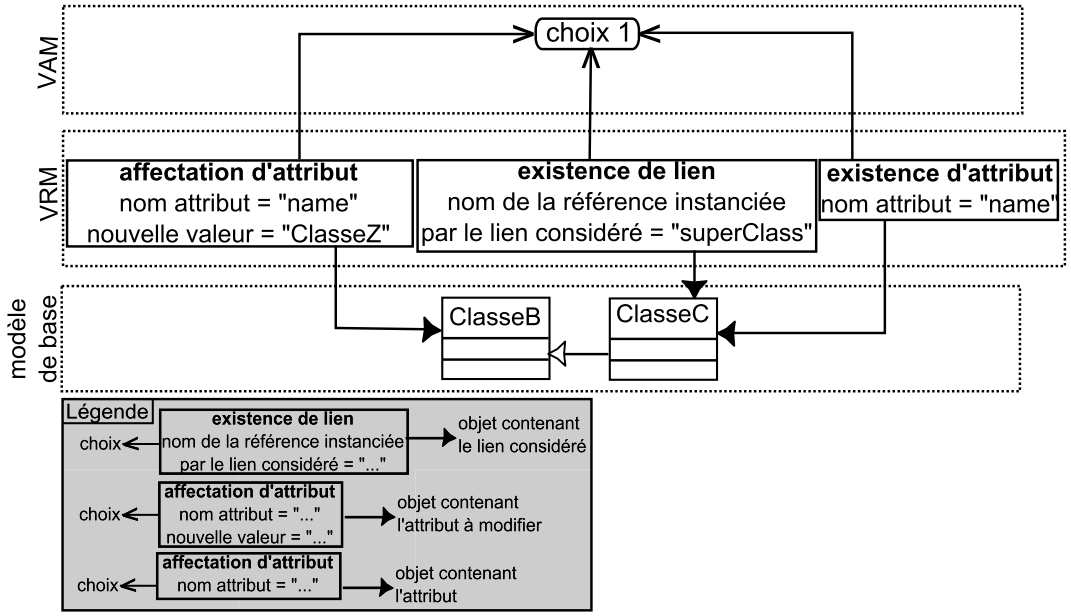


FIGURE 2.16 – Exemple de définition d'affectation d'attribut, d'existence de lien et d'existence d'attribut

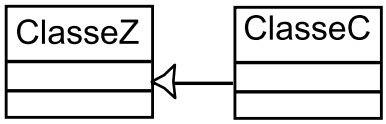


FIGURE 2.17 – Modèle résolu obtenu lorsque le choix de la figure 2.16 est sélectionné

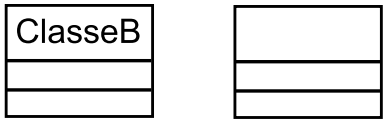


FIGURE 2.18 – Modèle résolu obtenu lorsque le choix de la figure 2.16 n'est pas sélectionné

la variabilité sur le nombre d'instances d'un fragment du modèle de base, tandis que l'instance de variabilité permet de déterminer ce nombre d'instances. Des exemples de VAM, VRM et modèle de base sont présentés en figure 2.19 afin d'illustrer une substitution de fragment. Si une instance de variabilité résout le classificateur de variabilité *mon classificateur de variabilité*, alors le fragment constitué des classes *ClasseB* et *ClasseC* est remplacé par le fragment constitué des classes *ClasseD* et *ClasseE* (cf. figure 2.20). Ce dernier est dupliqué si deux instances de variabilité résolvent *mon classificateur de variabilité* (cf. figure 2.21). Aucune substitution de fragment n'a lieu si aucune instance de variabilité ne résout *mon classificateur de variabilité*.

Un autre type de point de variation introduit par CVL est le point de variation

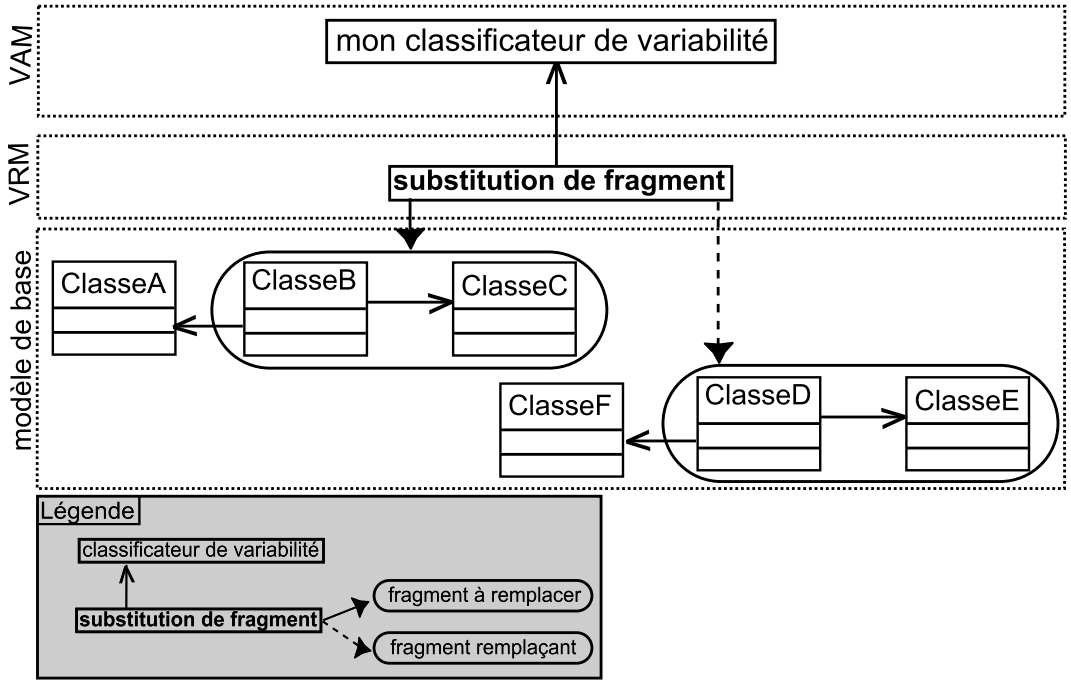


FIGURE 2.19 – Exemple de VAM, VRM et modèle de base pour une substitution de fragment

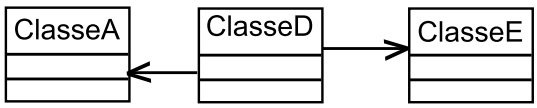


FIGURE 2.20 – Modèle résolu obtenu lorsque le classificateur de variabilité de la figure 2.19 est résolu par une seule instance de variabilité

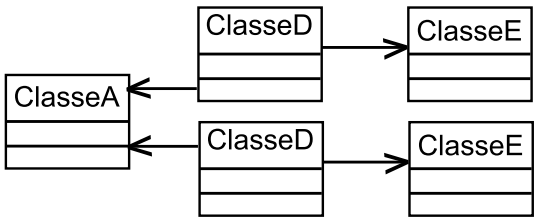


FIGURE 2.21 – Modèle résolu obtenu lorsque le classificateur de variabilité de la figure 2.19 est résolu par deux instances de variabilité

composite. Ce type de point de variation manipule des *conteneurs*, qui sont des objets du modèle de base contenant d'autres éléments de modèle, et sur lesquels de la variabilité est spécifiée. Comme illustré par la figure 2.22, le point de variation composite permet de cloner un conteneur (le paquetage *paquetage configurable* sur la figure), de résoudre la variabilité de chaque clone (paquetages *une configuration du paquetage configurable* et

une autre configuration du paquetage configurable sur la figure), et de rediriger vers ces clones les liens des éléments du modèle de base qui les utilisent (paquetages *paquetage 1* et *paquetage 2* sur la figure). Ce type de point de variation est utile lorsqu'un conteneur est utilisé plusieurs fois dans un même modèle, mais que les différentes utilisations de cet objet requièrent des résolutions différentes de sa variabilité. CVL définit également un type de spécification de variabilité ainsi qu'un type de résolution de spécification de variabilité spécifiques à l'utilisation des points de variation composites, à savoir la *spécification de variabilité composite* et la *configuration de variabilité*. La spécification de variabilité composite, qui est un conteneur pour d'autres spécifications de variabilité, permet de spécifier de la variabilité au niveau du contenu d'un élément du modèle de base. La configuration de variabilité permet quant à elle de résoudre cette variabilité.

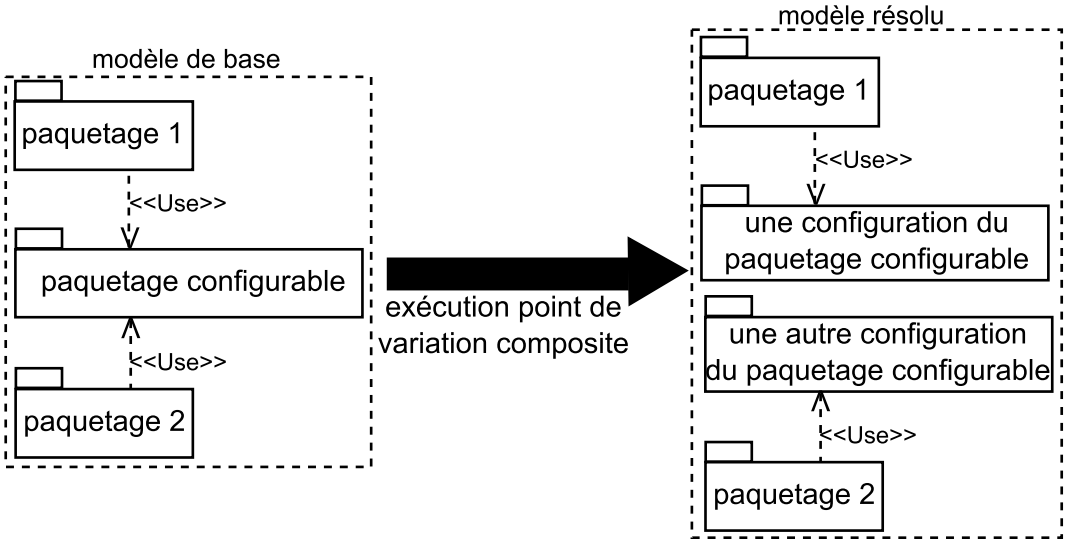


FIGURE 2.22 – Illustration du principe du point de variation composite

Enfin, CVL permet l'expression de contraintes transversales, c'est-à-dire de contraintes sur la résolution de spécifications de variabilité qui n'ont pas de relation de parent à enfant. Pour ce faire, CVL fournit le moyen d'exprimer des contraintes en utilisant la logique du premier ordre (incluant les quantificateurs universel et existentiel), ainsi que des contraintes arithmétiques.

2.4 Synthèse

Nous avons présenté dans ce chapitre les différents champs de recherche sur lesquels nous nous appuyons dans cette thèse, à savoir l'IDM, les processus de développement logiciel et l'ingénierie des lignes de produits logiciels. Il est à noter que ces trois champs sont également très dynamiques dans le domaine de la recherche puisqu'ils ont leurs propres conférences (*International Conference on Model Driven Engineering Languages and Systems*, *International Conference on Software and Systems Processes*

et *International Software Product Line Conference*) et qu'ils sont largement utilisés dans l'industrie [Nor08, HWRK11, Ins13]. Nous avons également présenté un langage de modélisation de processus de développement logiciel, SPEM 2.0, ainsi que CVL, un langage de modélisation permettant de spécifier et de résoudre de la variabilité sur des modèles.

Dans cette thèse, nous proposons de lier des automatisations de TMR (Tâches Manuelles Récurrentes) à des processus de développement logiciel. Nous utilisons ensuite ce lien pour savoir :

- quelles automatisations de TMR réutiliser pour un projet donné (en nous appuyant sur la réutilisation des processus de développement logiciel en fonction des exigences des projets),
- à quels moments d'un projet les réutiliser
- et pour identifier les différents cas d'utilisation de chaque automatisation de TMR et ainsi créer des automatisations de TMR qui soient réutilisables à travers ces différents cas d'utilisation.

L'IDM nous offre pour ce faire deux langages de modélisation spécifiques à deux préoccupations distinctes : SPEM 2.0 et CVL. SPEM 2.0 nous permet de définir les processus de développement logiciel et nous l'avons choisi car c'est un standard de l'OMG. Il est cependant tout à fait possible d'utiliser un autre langage de modélisation de processus de développement logiciel pour mettre en œuvre l'approche proposée dans cette thèse, en fonction des besoins d'un groupe d'utilisateurs. CVL nous permet de réutiliser les processus de développement logiciel. Il a l'avantage d'appliquer les principes de l'ingénierie des lignes de produits de logiciel et de pouvoir être utilisé quel que soit le langage de modélisation de processus utilisé pour peu qu'il soit conforme au MOF.

L'IDM nous offre de plus des outils (par exemple EMF) permettant l'implémentation de SPEM 2.0 et de CVL et la création de nouveaux langages de modélisation, tel qu'un langage spécifique aux automatisations de TMR. Enfin, l'utilisation d'un même métamétamodèle pour différents langages de modélisation permet de créer des liens entre ces langages. Il nous est ainsi possible de tisser les liens nécessaires dans le contexte de cette thèse, à savoir entre gestion de la variabilité et processus et entre processus et automatisations de TMR.

Chapitre 3

État de l'art : de la réutilisation d'automatisations de TMR à la réutilisation de processus

L'automatisation des TMR (Tâches Manuelles Récurrentes) liées à l'utilisation des outils de développement logiciel n'a qu'un intérêt limité si les mêmes automatisations de TMR sont re-développées pour chaque projet. La réutilisation d'automatisations de TMR amène cependant trois difficultés principales. La première est de savoir pour quels projets réutiliser une automatisation de TMR. En effet, une même automatisation de TMR peut être utile pour des projets différents, mais elle n'est pas forcément utile pour tous les projets. De plus, comme une automatisation de TMR peut avoir des dépendances avec d'autres tâches d'un projet, la deuxième difficulté est de déterminer à quels moments d'un projet utiliser une automatisation de TMR. La troisième difficulté est de s'assurer qu'une automatisation de TMR utile pour des projets différents, ou utile à des moments différents d'un même projet, soit bien réutilisable à travers ses différents cas d'utilisation. La difficulté est donc d'identifier le bon niveau de généricité d'une automatisation de TMR.

Une méthodologie est proposée par Sadovykh et Abherve [SA09] afin de supporter l'exécution de processus SPEM. Elle consiste à définir un processus SPEM et à le transformer en un processus BPEL [OAS07] afin de l'exécuter. L'apport principal de cette méthodologie est que le processus BPEL obtenu par transformation est directement exécutable. Pour ce faire, la méthodologie consiste à assigner au processus SPEM, avant sa transformation, les informations nécessaires pour le rendre exécutable (ressources humaines concrètes associées aux rôles, nombre des itérations et durée des tâches). Durant l'exécution du processus BPEL, les TMR sont automatisées à l'aide de services web, qui sont reliés aux unités de travail du processus BPEL et sont appelés au moment de l'exécution de ces unités de travail. Le lien entre les services web (similaires à des automatisations de TMR) et les unités de travail du processus qu'ils automatisent permet donc de savoir quand lancer ces services web au cours d'un projet. Mais la limite de cette méthodologie est qu'elle ne propose pas de support à la sélection des

automatisations de TMR à réutiliser pour un projet donné.

Si des automatisations de TMR sont liées à des processus de développement logiciel, alors réutiliser un processus de développement logiciel pour un projet donné permet de connaître les automatisations de TMR à réutiliser pour ce projet. Or, réutiliser les processus de développement logiciel est un exercice difficile. En effet, la variabilité au niveau des exigences des projets est la source d'un nombre de processus trop élevé pour qu'un humain puisse tous les connaître et savoir quand les réutiliser [Rom05, CDCC+13]. De nombreuses approches sont donc apparues afin d'apporter un support à la réutilisation des processus de développement logiciel.

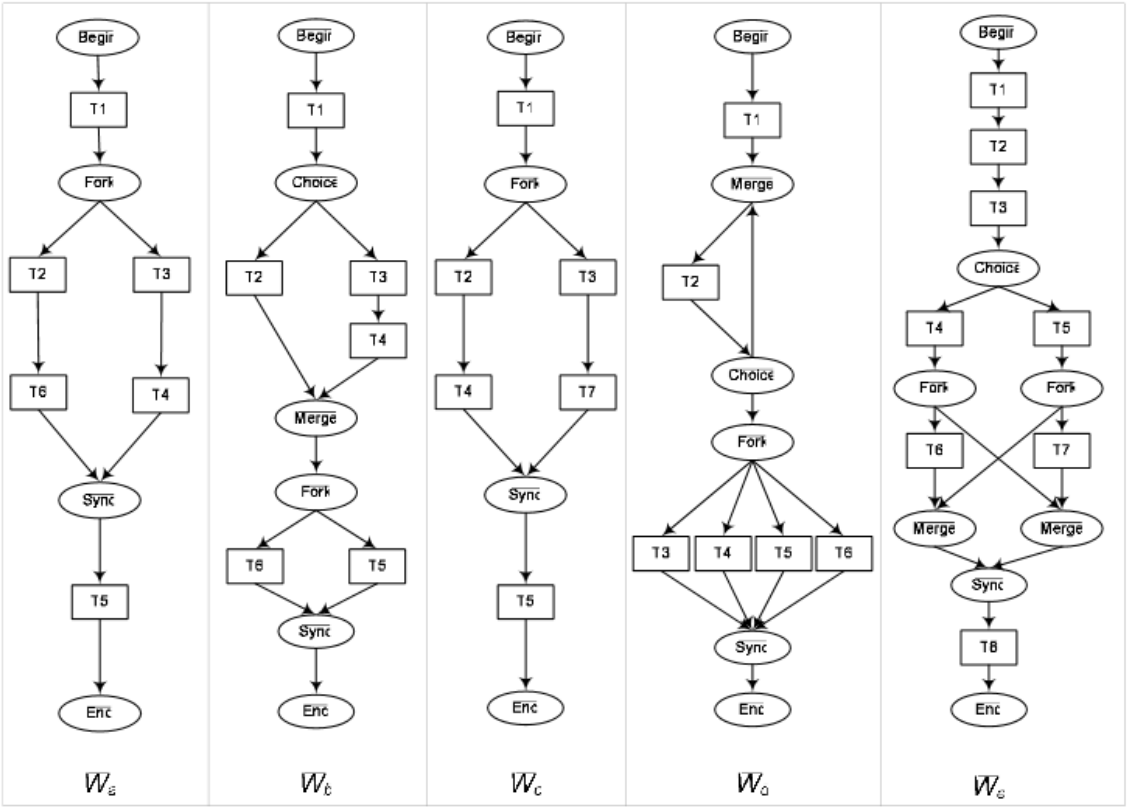


FIGURE 3.1 – Représentation en extension des processus d'une famille [LS07]

Dans le cadre de cette thèse, nous fixons certaines exigences qu'une telle approche doit satisfaire. Premièrement, elle doit permettre de définir un ensemble de processus et de dériver automatiquement un processus de cet ensemble, afin de permettre la réutilisation des processus. Deuxièmement, l'approche doit permettre de définir les différents processus à réutiliser en intention, c'est-à-dire en factorisant leurs parties communes, afin d'en faciliter la maintenance et de permettre leur réutilisation [RMvdT09, HBR10, KY12]. Plus précisément, la définition en intention de processus consiste à ne définir qu'une seule fois une partie commune à plusieurs processus et à y faire référence pour l'utiliser. À l'inverse, la définition en extension de processus

consiste à les définir entièrement et indépendamment les uns des autres, en dupliquant leurs parties communes d'un processus à l'autre (mais pas nécessairement au sein d'un même processus puisque les flots de contrôle sont souvent utilisés pour retourner à une partie d'un processus). La figure 3.1, issue de [LS07], illustre la représentation en extension de différents processus d'une même famille. On remarque que ces processus partagent des parties communes qui sont dupliquées (par exemple la tâche *T1*). Cela pose des problèmes de maintenance car lorsqu'une de ces parties évolue, elle doit être mise à jour dans tous les processus auxquels elle appartient, ce qui est source d'erreurs et coûteux en temps. Cela limite de plus la réutilisation de fragments de processus. Troisièmement, une approche supportant la réutilisation des processus doit pouvoir s'appliquer quel que soit le formalisme utilisé pour définir les processus, afin que les utilisateurs de cette approche puissent utiliser le formalisme qui satisfait le mieux leurs exigences [Mee10, WKK+11, KY12]. Cela permet à un groupe d'utilisateurs de pouvoir changer de formalisme de définition de processus, mais cela permet également que l'approche puisse être utilisable par des groupes d'utilisateurs utilisant des formalismes différents.

En plus de nous intéresser aux approches traitant de la réutilisation des processus de développement logiciel, nous nous intéressons aux approches traitant de la réutilisation des processus métiers, où un processus métier correspond à l'ensemble des tâches permettant de créer un résultat de valeur pour un client [Ham96]. En effet, un processus de développement logiciel est également un processus métier [Ben07]. De la même manière, comme un *workflow*, c'est-à-dire une automatisation partielle ou totale d'un processus métier [(WF99], peut être considéré comme un processus métier [Ben07, LRvdADM13], et que les diagrammes d'activités UML sont parfois utilisés pour capturer les processus métiers [LRvdADM13], nous incluons dans cet état de l'art les approches traitant de la réutilisation des *workflows* et des diagrammes d'activités.

Nous présentons les approches supportant la réutilisation de processus dans la suite de ce chapitre et nous les évaluons en fonction des exigences précédemment définies. La section 3.1 présente des approches apportant un support à la réutilisation des processus sans traiter de la spécification de leur variabilité. La section 3.2 porte sur les approches s'appuyant sur l'ingénierie des lignes de processus [Rom05] afin de réutiliser les processus, où l'ingénierie des lignes de processus est une activité similaire à l'ingénierie des lignes de produits mais où le produit est un processus. Nous faisons la synthèse de ces approches en section 3.3.

3.1 Support à la réutilisation des processus sans spécification de leur variabilité

Nous présentons dans cette section des approches offrant des mécanismes pour supporter la réutilisation des processus sans pour autant permettre de spécifier leur variabilité. La section 3.1.1 traite des approches permettant d'identifier le processus correspondant le mieux à un projet. La section 3.1.2 présente des approches permettant de réutiliser des processus définis en extension. Enfin, la section 3.1.3 aborde les patrons

de processus.

3.1.1 Identification du processus correspondant à un projet

Les approches suivantes permettent d'identifier à quels types de projets correspondent quels types de processus, afin de faciliter le choix d'un processus pour un projet donné. Elles sont toutes assez abstraites pour pouvoir être appliquées quel que soit le formalisme de définition de processus utilisé.

L'approche Promote (***Pro**cess **Mo**dels **T**axonomy for **E**nlightening choices*) [CDCC+13] définit un ensemble de caractéristiques de processus de développement logiciel et propose une classification de processus existants en fonction de ces caractéristiques. Les processus sont caractérisés selon 6 axes principaux : i) le cycle, c'est-à-dire la granularité des itérations, la durée du cycle de vie, les incréments, etc., ii) la collaboration, c'est-à-dire les différents types de relations qui existent entre les intervenants sur un projet iii) les artéfacts, c'est-à-dire les types de livrables qui doivent être produits, iv) l'utilisation recommandée, c'est-à-dire dans quels contextes un processus est le plus adapté, v) le niveau de maturité du processus, et vi) la flexibilité, c'est-à-dire la capacité du processus à être adapté. Cependant, Promote ne propose pas de mécanisme permettant d'obtenir un processus correspondant à un projet dans un formalisme permettant son exécution. D'autre part, la problématique de la définition d'une famille de processus en intention est en dehors des objectifs de cette approche.

D'autres approches, en plus de proposer des caractéristiques pour définir les processus et les projets, proposent un support à la sélection du processus correspondant le mieux à un projet. Ainsi, l'approche de Pérez et al. [PEM95] propose une méthodologie permettant d'évaluer l'efficacité d'un processus pour un projet donnée, en fonction de leurs caractéristiques respectives. L'approche d'Alexander et Davis [AD91] fournit quant à elle une méthode permettant de déterminer le processus qui correspond le mieux à un projet donné, toujours en fonction des critères des processus et de ce projet. L'approche de Sharon et al. [SdSSB+10] propose un *framework* supportant la sélection du processus correspondant le mieux à un projet donné. Mais ces approches ne traitent pas de la définition d'une famille de processus et de la dérivation d'un processus de cette famille en fonction des exigences des projets.

Comme résumé dans le tableau 3.1, les approches présentées ici, bien qu'indépendantes du formalisme utilisé pour définir les processus, ne fournissent pas tous les mécanismes nécessaires à la réutilisation des processus de développement logiciel. La définition en intention d'un ensemble de processus n'est de plus pas traitée par ces approches.

Approche	Concepts proposés	Apports réutilisation processus	Limites réutilisation processus	Déf. des processus en intention	Indépendante du formalisme pour définir les processus
Promote [CDCC+13]	Caractéristiques de processus et classification de processus en fct. de ces caractéristiques	Identification du processus correspondant à un projet	Obtention d'un processus exécutable	non	oui
Pérez et al. [PEM95]	Méthodologie pour évaluer l'efficacité d'un processus pour un projet donné	Identification du processus correspondant à un projet	Déf. d'une famille de processus et dérivation d'un processus	non	oui
Alexander et Davis [AD91]	Méthode pour déterminer le processus qui correspond le mieux à un projet	Identification du processus correspondant à un projet	Déf. d'une famille de processus et dérivation d'un processus	non	oui
Sharon et al. [SdSSB+10]	*Framework* supportant la sélection du processus correspondant le mieux à un projet	Identification du processus correspondant à un projet	Déf. d'une famille de processus et dérivation d'un processus	non	oui

TABLE 3.1 – Évaluation des approches permettant d'identifier le processus correspondant le mieux à un projet

3.1.2 Réutilisation de processus définis en extension

D'autres approches permettent quant à elles de définir un ensemble de processus et de réutiliser des processus de cet ensemble. L'approche de Lu et Sadiq [LS07] permet l'extraction d'un processus depuis un ensemble de processus capturés dans un dépôt, en fonction des caractéristiques de ce processus. Elle définit pour ce faire une *mesure de similarité*, qui permet de déterminer quel est le processus du dépôt qui est le plus proche d'un ensemble de caractéristiques. Cette approche est cependant dépendante du formalisme utilisé pour définir les processus. En effet, la définition de la mesure de similarité dépend de la sémantique des éléments à comparer. De plus, l'approche est définie pour un formalisme de processus particulier et devrait donc être adaptée pour pouvoir être utilisée avec un formalisme différent.

L'approche de Song et Osterweil [SO98] applique le principe de la programmation des processus [Ost87] à la méthodologie de conception logicielle BOOD (*Booch Object Oriented Design Methodology*) [Boo91]. Le principe de la programmation des processus suggère que les processus soient considérés comme des logiciels et soient donc conçus, développés et exécutés comme tels. Il est assez générique pour pouvoir être utilisé quel que soit le langage de modélisation de processus. BOOD est considérée comme étant l'architecture d'un processus de conception. Des processus de conception plus détaillés sont élaborés à partir de cette méthodologie puis exécutés, à l'aide du prototype Debus-Booch. Celui-ci permet entre autres de sélectionner un processus de conception logiciel depuis une famille de tels processus. Il apporte de plus un support à cette sélection en fournissant des informations sur la nature des différents processus et des étapes qui les composent.

Bien que permettant la réutilisation des processus, les deux approches précédentes ne proposent pas de mécanismes permettant de définir un ensemble de processus en intention, comme résumé dans le tableau 3.2. À défaut d'un tel mécanisme, les processus sont donc définis en extension.

3.1.3 Les patrons de processus

Un patron de processus, ou composant de processus, est un fragment de processus réutilisable [GLKD98, ABC96, Amb99]. De nombreuses approches proposent des patrons de processus (par exemple [VDATHKB03, RHEdA05, KR10, BnCCG11, KYSR09, IMCH+07]). Ceux-ci peuvent être définis à des niveaux d'abstraction différents. Des patrons généraux devront donc être adaptés à un contexte particulier pour pouvoir être réutilisés, tandis que des patrons plus détaillés pourront être réutilisés tels quels [GLKD98, ABC96, GM05, Amb99]. Certains langages de modélisation de processus, comme SPEM 2.0 ou Little-JIL [Wis06], permettent quant à eux de définir des patrons de processus et de les intégrer à un processus. D'autre part, plusieurs approches proposent de définir des processus par réutilisation et composition de patrons de processus [GLKD98, ABC96, Zhu03, PS05, BBM05]. Elles fournissent les mécanismes nécessaires à la définition et à la composition de ces patrons. Mais, dans toutes les précédentes approches, la sélection et la composition de patrons de processus, ou

Approche	Concepts proposés	Apports réutilisation processus	Limites réutilisation processus	Déf. des processus en intention	Indépendante du formalisme pour définir les processus
Lu et Sadiq [LS07]	Extraction d'un dépôt du processus le plus proche d'un ensemble de caractéristiques	Définition d'un ensemble de processus et réutilisation des processus de cet ensemble	aucune	non	non
Song et Osterweil [SO98]	Application du principe de la programmation des processus à BOOD	Définition d'un ensemble de processus et réutilisation des processus de cet ensemble	aucune	non	oui

TABLE 3.2 – Évaluation des approches de Lu et Sadiq [LS07] et Song et Osterweil [SO98]

leur intégration à un processus, restent manuelles.

D'autres approches apportent donc un support à la réutilisation de patrons de processus. L'outil *Intelligent Workflow Designer* [IMCH+07] permet la modélisation de processus métiers basée sur la réutilisation de patrons de *workflows*. Au moment de la modélisation d'un processus, il analyse ce processus et propose des patrons à appliquer, parmi un ensemble de patrons prédéfinis. Mais la modélisation d'un processus reste en grande partie manuelle. Un processus doit d'ailleurs être modélisé manuellement une première fois avant d'être analysé en vue de détecter les patrons qui pourraient être appliqués. L'outil ne permet pas de définir un ensemble de processus en intention et l'indépendance vis-à-vis des langages de modélisation de processus n'est pas traitée.

L'approche *Design by Selection* [ASKW11] permet également la réutilisation de composants de processus durant la modélisation d'un processus. Un processus est dans un premier modélisé manuellement et l'approche permet de définir manuellement les parties de ce processus pour lesquelles un composant doit être réutilisé. Ces parties sont alors utilisées pour faire des requêtes sur un dépôt de composants de processus, afin de récupérer un ensemble de composants pouvant correspondre à une partie de processus. La sélection d'un composant parmi l'ensemble retourné est elle aussi manuelle, même si l'approche offre un support à cette sélection en classant les composants proposés en fonction de leur pertinence. Là encore une partie de la définition des processus reste manuelle et leur réutilisation n'est donc pas complètement automatisée. L'approche est assez abstraite pour pouvoir être appliquée indépendamment d'un langage de modélisation de processus. Mais son implémentation dépend quant à elle d'un tel langage. Cette approche ne traite pas non plus de la définition de processus

en intention.

L'outil CAT (*Composition Analysis Tool*) [KSG04] assiste un utilisateur dans la définition de *workflows* basée sur la réutilisation de composants. CAT analyse un *workflow* en cours de définition et suggère des actions possibles à la personne en charge de cette définition (par exemple des ajouts ou des suppressions de composants) et s'assure que le *workflow* produit est correct (par exemple en vérifiant que des flots de contrôle de sont pas inconsistants avec la description d'un composant). La définition de processus reste là encore en grande partie manuelle et la spécification de processus en intention ainsi que l'indépendance vis-à-vis des langages de modélisation de processus ne sont pas abordées.

Aldin [Ald10] propose une méthodologie pour l'extraction de patrons de processus métiers depuis un ensemble de processus et la réutilisation de ces patrons lors de la modélisation d'un nouveau processus métier. Un patron de processus est sélectionné en fonction des exigences correspondant à un problème métier. Il est ensuite adapté à un problème métier spécifique puis intégré au processus correspondant à ce problème métier. Cette méthodologie est assez générique pour pouvoir être appliquée quel que soit le langage de modélisation de processus utilisé. En revanche, rien n'est proposé afin d'automatiser la réutilisation des patrons de processus. La spécification en intention d'un ensemble de processus n'est pas non plus abordée.

Comme résumé dans le tableau 3.3, plusieurs approches apportent un support à la réutilisation de patrons de processus, mais la définition d'un processus reste en grande partie manuelle. Ces approches ne permettent pas de capturer un ensemble de processus et d'en dériver automatiquement un processus. D'autres approches permettent quant à elles de réutiliser automatiquement un processus dont la définition s'appuie sur des patrons de processus [AB12, Ist13, BDKK04, FPLPdL01]. Nous les présentons plus en détails dans la section 3.2.

3.2 Ingénierie des lignes de processus

D'autres approches s'appuient sur l'ingénierie des lignes de processus [Rom05] afin de réutiliser les processus. L'ingénierie des lignes de processus est une activité similaire à l'ingénierie des lignes de produits, mais où le produit est un processus. Les approches existantes définissent une ligne de processus en spécifiant les différents processus d'une famille ainsi que leurs parties variables et communes [Ter09]. Une ligne de processus est donc définie en intention et non en extension. Ainsi, lorsqu'une partie commune à plusieurs processus évolue, celle-ci n'est mise à jour qu'une seule fois. Il est possible de réutiliser un processus d'une famille en le dérivant de la ligne de processus. Les approches existantes s'appuient sur l'IDM afin de définir une ligne de processus et d'en dériver un processus. Nous détaillons ces approches dans la suite de cette section.

Approche	Concepts proposés	Apports réutilisation processus	Limites réutilisation processus	Déf. des processus en intention	Indépendante du formalisme pour définir les processus
Intelligent Workflow Designer [IMCH+07]	proposition de patrons à réutiliser	réutilisation de patrons de *workflows*	déf. de processus en partie manuelle	non	non
Design by Selection [ASKW11]	proposition de patrons à réutiliser	réutilisation de composants de processus	déf. de processus en partie manuelle	non	non
CAT [KSG04]	proposition de patrons à réutiliser	réutilisation de composants de *workflows*	déf. de processus en partie manuelle	non	non
Aldin [Ald10]	méthodologie pour la découverte et la réutilisation de patrons de processus	réutilisation de patrons de processus	déf. de processus et réutilisation de patrons manuelles	non	oui

TABLE 3.3 – Évaluation des approches apportant un support à la réutilisation de patrons de processus, mais sans permettre la réutilisation automatique de processus

3.2.1 Utilisation des mécanismes d'un langage

Deux approches, d'Alegría et al. [HABQO11] et d'Alegría et Bastarrica [AB12], gèrent la variabilité des processus de développement logiciel en s'appuyant sur les mécanismes de variabilité fournis par SPEM 2.0. SPEM 2.0 offre en effet des mécanismes permettant de spécifier qu'un élément de processus est optionnel, d'ajouter des propriétés à un élément ou d'étendre ou de remplacer les propriétés d'un élément. Il permet également de définir des patrons de processus, que l'approche d'Alegría et Bastarrica permet de réutiliser automatiquement lors de la dérivation d'un processus. Cependant, ces mécanismes sont spécifiés directement dans le métamodèle de SPEM 2.0 et aucun moyen permettant de les découpler de ce métamodèle n'est proposé. En conséquence, réutiliser ces mécanismes avec un autre langage de modélisation de processus implique, en plus de devoir adapter ces mécanismes, de devoir modifier le métamodèle de cet autre langage.

3.2.2 Extension d'un langage de modélisation de processus

Les approches de cette catégorie étendent un langage de modélisation de processus existant avec des concepts permettant de gérer la variabilité. Cependant, la plupart d'entre elles ne propose pas de mécanisme permettant d'appliquer ces concepts à un langage sans le modifier. Celles qui le font proposent quant à elles des concepts qui doivent être adaptés pour pouvoir être réutilisés avec des langages de modélisation de processus autres que celui pour lequel ils ont été initialement définis.

3.2.2.1 Utilisation de processus agrégats

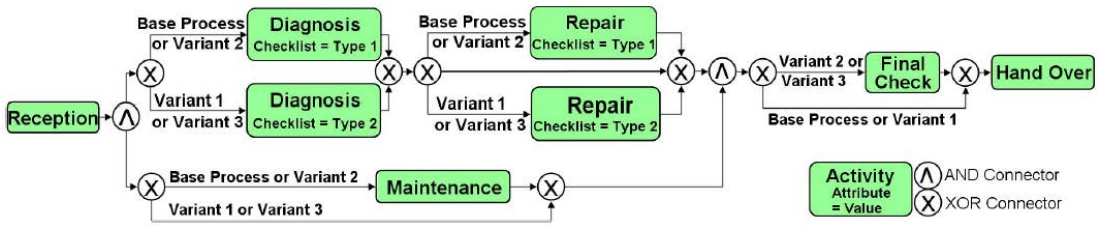


FIGURE 3.2 – Un exemple de modèle de processus agrégat [HBR10]

Afin de gérer la variabilité des processus, certaines approches s'appuient sur des modèles de processus *agrégats* [RMvdT09]. Un processus agrégat capture les différents processus d'une famille en les combinat dans un même et unique processus à l'aide de mécanismes spécifiques à la modélisation d'un flot d'unités de travail (nœuds de décision, de jointure, de parallélisation, flots de contrôle, etc.) [KL07, RvdA07]. Par exemple, la figure 3.2 illustre un processus agrégat qui capture les différents processus d'une famille en utilisant des nœuds de décision. Néanmoins, un simple modèle de processus agrégat n'offre pas de support à la réutilisation des processus [GvdAJV07, RvdA07, HBR10]. Par exemple, il n'est pas possible de distinguer les nœuds de décision représentant des choix qui doivent être faits au moment de l'exécution d'un processus de ceux qui dénotent des processus différents [HBR10]. Il n'est pas non plus possible de distinguer les éléments de processus optionnels [RvdA07]. Les approches existantes proposent donc d'étendre un langage de modélisation de processus avec des mécanismes permettant d'expliciter la variabilité dans un modèle de processus agrégat, et elles offrent un support à la sélection et à la dérivation d'un processus de ce processus agrégat. Selon les approches, un processus peut ainsi être dérivé par sélection ou omission d'éléments du processus agrégat, par adaptation des éléments du processus agrégat à des exigences particulières, par ajout d'éléments spécifiques à un cas particulier, ou par instanciation (c'est-à-dire en fournissant une implémentation à un concept abstrait). Aucun moyen n'est cependant proposé afin d'étendre un langage de modélisation de processus sans le modifier. Nous présentons ces approches dans la suite de cette partie, en détaillant plus particulièrement les extensions qu'elles proposent.

L'approche KobrA (*KOmponenten BasieRte Anwendungsentwicklung*, allemand pour

développement d'applications basé sur les composants) [ABM00] intègre l'ingénierie des lignes de produits logiciels et l'ingénierie logicielle basée sur les composants [Crn02]. Dans ce cadre, elle propose une notation afin de distinguer dans un processus agrégat les nœuds de décision dénotant des processus différents de ceux dénotant des choix à faire au moment de l'exécution d'un processus.

L'approche *Superimposed Variants* [CA05] s'appuie sur les BFM (*Basic Feature Model*) [KCH+90] afin de gérer la variabilité de modèles et permet de faire le lien entre les BFM et les modèles desquels ils spécifient la variabilité. Cette approche étend un langage de modélisation avec des mécanismes permettant d'associer à des éléments de modèle i) des informations déterminant dans quels cas ces éléments sont sélectionnés et ii) des méta-expressions, qui spécifient comment configurer les propriétés d'un élément en fonction de ses différents cas d'utilisation. Cette approche n'est pas spécifique à la gestion de la variabilité dans les processus mais est appliquée à des modèles de processus agrégats définis à l'aide de diagrammes d'activité UML 2.0.

D'autres approches permettent de configurer des modèles de processus référence, où un processus de référence est un processus général qui définit les bonnes pratiques recommandées dans un domaine particulier [FL03, Fra99]. Dans ces approches, les modèles de processus de référence sont définis comme des modèles agrégats, bien que rien n'impose qu'un modèle de référence soit un modèle agrégat [Tho05].

Ainsi, l'approche C-EPC (*Configurable Event-driven Process Chain*) [RvdA07] permet de configurer les processus de référence définis à l'aide du langage de modélisation de processus EPC [Sch00]. Celui-ci est étendu avec des mécanismes permettant de spécifier qu'une unité de travail est optionnelle, qu'un connecteur (c'est-à-dire un nœud de décision, de jointure ou de parallélisation) peut être remplacé par un connecteur qui restreint son comportement, ou qu'au moment de la dérivation un seul des flots de contrôle navigables à l'issue d'un nœud de décision pourra être sélectionné. C-EPC propose également d'étendre EPC avec des mécanismes permettant d'exprimer des contraintes entre les résolutions d'éléments de modèle variables (implication, exclusion, etc.).

L'approche C-iEPC (*Configurable integrated EPC*) [RDH+08, LRDtHM11] étend EPC avec les concepts de rôle et d'objet (où la notion d'objet est similaire à la notion de produit de travail) et avec des mécanismes permettant de spécifier que des rôles ou des objets sont optionnels, alternatifs, ainsi que le nombre minimal et maximal de rôles ou d'objets qui peuvent être sélectionnés.

L'approche aEPC (*Aggregate EPC*) [RMvdT09] permet de déterminer les cas d'utilisation des éléments d'un processus de référence. Pour ce faire, elle étend le langage de modélisation de processus utilisé avec des mécanismes permettant d'associer aux éléments d'un modèle de processus de référence des informations déterminant dans quels cas ces éléments sont sélectionnés.

L'approche *Configurative Process Modeling* [BDK07] étudie l'applicabilité des mécanismes utilisés pour configurer des modèles de processus de référence aux méthodes d'ingénierie logicielle. Dans ce contexte, elle propose d'étendre le formalisme utilisé pour définir une méthode logicielle avec des mécanismes permettant i) d'associer aux éléments d'une méthode des informations déterminant dans quels cas ces éléments

sont sélectionnés, ii) de définir qu'un élément d'une méthode est abstrait et doit être instancié au moment de la dérivation et iii) d'associer à un élément d'une méthode des informations aidant à l'adapter aux exigences d'un projet.

L'approche ADOM (*Application-based DOmain Modeling*) [RBSS09, RBSS10] permet de définir la variabilité d'un modèle de processus de référence, d'en dériver un processus spécifique et de vérifier qu'un processus spécifique appartient bien à la famille de processus capturée par le modèle de référence. Un processus peut être dérivé en adaptant les éléments du modèle de processus agrégat ou en lui ajoutant des éléments, en plus de pouvoir sélectionner ou omettre des éléments du processus agrégat. ADOM étend un langage de modélisation de processus avec les concepts d'*indicateur de multiplicité* et de *classificateur de modèle de référence*. Un indicateur de multiplicité permet de spécifier qu'un élément du modèle de processus de référence est optionnel, obligatoire, qu'il a des variantes et combien. Un classificateur de modèle de référence permet de spécifier à quel élément du modèle de processus de référence un élément d'un modèle de processus résolu correspond. Il permet de vérifier qu'un processus appartient bien à la famille de processus capturée par le modèle de processus de référence.

L'approche *Configurable Workflow* [GvdAJVLR08] se concentre sur la configuration de modèles de *workflows*, afin de permettre la production de modèles directement exécutables. Elle propose d'étendre les unités de travail d'un langage de modélisation de *workflows* avec des *ports* d'entrée et de sortie, où un port est un endroit connectant un flot de contrôle à une unité de travail. La configuration d'un modèle de processus de référence consiste à définir si un port est *activé*, *bloqué* ou *caché*. Cela permet de spécifier les parties d'un *workflow* qui peuvent être exécutées.

3.2.2.2 Modification d'un processus de base

Une limitation des approches qui s'appuient sur les modèles de processus agrégats est que lorsque les processus à capturer sont complexes et nombreux, alors le processus agrégat devient illisible et difficile à créer et à maintenir [HBR10, DB10]. Afin de faire face à cette difficulté, l'approche Provop (*PROcess Variants by OPtions*) [HBR10] propose de modéliser un processus de base (qui peut être un processus de référence, un processus d'une famille, le processus commun à tous les processus d'une famille, etc.) et de définir les opérations à appliquer à ce processus afin d'en dériver les différents processus d'une famille. Des *points d'ajustement* sont utilisés pour spécifier les parties du processus de base auxquelles des opérations sont appliquées. Cependant, aucun mécanisme n'est proposé pour définir des points d'ajustement sans modifier le langage de modélisation de processus utilisé.

L'approche proposée par Mosser et Blay-Fornarino [MBF13] supporte l'évolution automatique de modèles de processus métiers, c'est-à-dire la modification automatique de modèles de processus métiers afin qu'ils soient conformes aux changements apportés à un domaine métier. Cette approche supporte également la détection d'*interférences* entre les évolutions apportées à un processus métier, où une interférence est une interaction entre des évolutions qui engendre un comportement non souhaité d'une ou plusieurs de ces évolutions. Une évolution est apportée à un modèle de processus

métier en lui composant un fragment de processus. À cette fin, le métamodèle ADORE (*Activity meta-moDel supOrting oRchestration Evolution*) est proposé. Il permet de définir des processus métiers, des fragments de processus ainsi que les endroits d'un processus où ces fragments peuvent être intégrés. Mais aucun mécanisme n'est proposé afin de pouvoir définir des fragments de processus ainsi que leurs points d'intégration sans modifier le langage de modélisation de processus utilisé.

3.2.2.3 Spécification des points de variation et des variantes d'un processus

Plusieurs approches définissent une ligne de processus en modélisant le processus commun à tous les processus d'une famille et en spécifiant quelles sont les parties de ce processus qui varient (points de variation) et leurs différentes valeurs (variantes). Contrairement aux processus agrégats, les différents processus d'une famille ne sont pas tous intégrées dans un même processus en utilisant les mécanismes spécifiques à la modélisation d'un flot d'unités de travail. La figure 3.3 illustre un exemple de définition de points de variation et de variantes sur un modèle de processus. Elle illustre que l'unité de travail *P2* est en fait un point de variation ayant pour variantes les unités de travail *A1* à *An*. Un processus d'une famille est dérivé en remplaçant chaque point de variation par une variante, sélectionnée en fonction des exigences d'un projet. Les approches existantes étendent donc un langage de modélisation de processus avec des mécanismes permettant de définir des points de variation et des variantes.

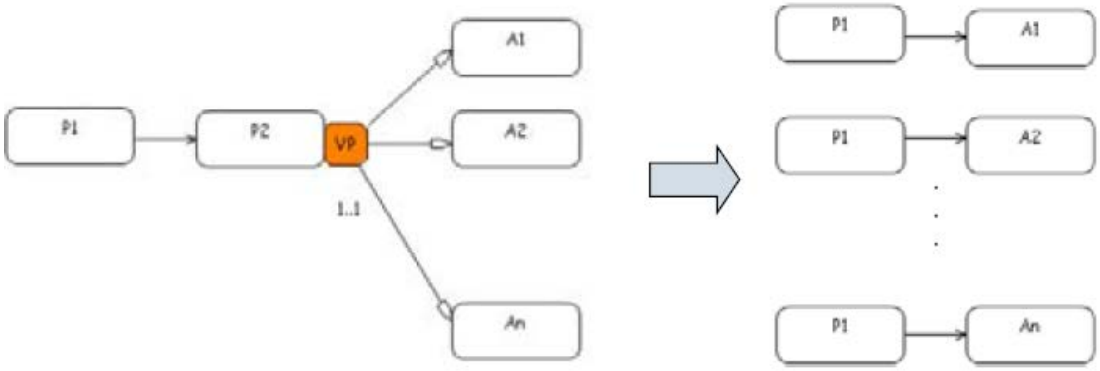


FIGURE 3.3 – Exemple de définition de points de variation et de variantes sur un modèle de processus [MHY08]

Ainsi, l'approche vSPEM [MRGP08] étend SPEM 2.0 avec des mécanismes de variabilité évalués empiriquement comme plus compréhensibles que ceux proposés par SPEM 2.0 [MRGPM11]. Ces mécanismes permettent de définir que des activités, des produits de travail, des rôles ou des tâches sont des points de variation. Ils permettent également de définir les variantes associées à chaque point de variation ainsi que des relations de dépendance (inclusion et exclusion) entre points de variation et variantes.

L'approche de Kulkarni et Barat [KB10] étend le langage de modélisation de processus BPMN (*Business Process Model and Notation*) [OMG11a] afin de gérer la variabilité

dans les modèles de processus BPMN. L'extension proposée permet de spécifier qu'une activité ou qu'un évènement sont des points de variation, quelles sont leurs variantes, et de sélectionner les variantes à affecter aux points de variation.

L'approche de Nguyen et al. [NCH11] gère la variabilité dans des services définis par des processus métiers. Elle s'attache en particulier à gérer les dépendances de variabilité entre un service et les services qui le compose. Afin de gérer la variabilité de services, elle étend BPMN avec des concepts permettant de définir des points de variation sur le séquencement d'activités, ainsi que sur les données et messages échangés par ces activités. Ces concepts permettent également de spécifier les variantes d'un point de variation ainsi que le nombre minimum et maximum de variantes qui peuvent être sélectionnées pour chaque point de variation.

Trois approches, PESOA (*Process Family Engineering in Service-Oriented Applications*) [PSWW05], BPFM (*Business Process Family Model*) [MHY08] et celle proposée par Ripon et al. [RTM10], s'appuient sur les stéréotypes UML afin de spécifier la variabilité de processus capturés dans des diagrammes d'activités UML. L'approche PESOA identifie les mécanismes de variabilité pertinents pour la gestion de la variabilité dans les processus, parmi un ensemble de mécanismes existants (paramétrage, extension, remplacement, omission, ajout, etc.), et les applique aux processus. Dans ce contexte, elle propose des stéréotypes permettant de spécifier qu'une unité de travail est un point de variation, quelles sont ses variantes, si une variante peut être utilisée par défaut, si des variantes sont mutuellement exclusives et si une unité de travail est optionnelle. L'approche PESOA propose également d'utiliser les mêmes stéréotypes pour spécifier la variabilité de processus modélisés avec BPMN. Pour cela, elle propose d'étendre BPMN avec le concept de stéréotype.

L'approche BPFM consiste à analyser la variabilité d'un processus métier à différents niveaux d'abstraction afin d'identifier les points de variation et les variantes de ce processus. Afin de spécifier ces derniers, elle propose des stéréotypes permettant de définir qu'une activité est optionnelle ou obligatoire, qu'elle a des variantes, le nombre minimum et maximum de variantes d'une activité qui peuvent être sélectionnées, ainsi que les dépendances entre les points de variation et les variantes. Ces stéréotypes permettent également de spécifier de la variabilité au niveau des flots de contrôle. Il est aussi possible de spécifier si de nouvelles variantes peuvent être définies au moment de la résolution.

L'approche proposée par Ripon et al. se concentre sur la gestion de la variabilité dans les modèles de domaine, qui définissent les aspects conceptuels d'un système. Elle propose un stéréotype afin de spécifier les éléments d'un modèle UML qui ont des variantes. Les éléments marqués avec ce stéréotype sont également marqués avec des valeurs taguées, afin d'identifier quelles sont leurs variantes et quelles décisions un ingénieur doit prendre pour résoudre la variabilité de ces éléments. Cette approche est appliquée sur des diagrammes d'activité UML.

Deux approches, l'une proposée par Ternité [Ter09] et l'autre par Ciuksys et Caplinskas [CC07], fournissent un métamodèle permettant de définir une ligne de processus. Le métamodèle proposé par Ternité permet de définir que des éléments de processus sont optionnels, obligatoires, ou alternatifs. Le métamodèle proposé par

Ciuksys et Caplinskas permet de spécifier de la variabilité uniquement sur des activités. Il permet de définir qu'une activité est un point de variation, quelles sont ses variantes ainsi que les dépendances entre ses variantes. Les deux métamodèles offrent également des concepts permettant de résoudre la variabilité et de spécifier les opérations à réaliser sur la ligne de processus afin d'en dériver un processus. Mais afin d'être concrètement utilisés, ces métamodèles doivent être spécialisés en fonction du langage de modélisation de processus. Plus précisément, les éléments d'un langage de modélisation de processus sur lesquels spécifier de la variabilité doivent hériter de certains des concepts proposés par ces métamodèles.

L'approche proposée par Acher et al. [ACG+12] permet de gérer la variabilité des services qui composent un *workflow*, où un service est un composant qui automatise une activité de ce *workflow*. Des BFM sont utilisés afin de spécifier la variabilité des services. Afin de lier ces BFM aux services dont ils spécifient la variabilité, chaque service est augmenté avec un identifiant unique.

Toutes les approches précédentes sont cependant dépendantes du langage de modélisation de processus utilisé. En effet, afin d'être appliquées, elles nécessitent toutes d'étendre ce langage, mais aucune approche ne propose de mécanisme permettant de réaliser cette extension sans modifier ce langage.

L'approche ABIS (*Adaptive Business Process Modeling in the Internet of Services*) [WKK+11] introduit quant à elle des constructions pour gérer la variabilité dans des processus BPMN, mais sans modifier le métamodèle BPMN. En effet, ces constructions sont instanciées dans un fichier XML différent du fichier dans lequel sont définis les processus BPMN. Ces constructions permettent de définir de la variabilité au niveau d'une activité BPMN ou au niveau d'un attribut d'un élément de modèle BPMN. Des fragments de processus spécifient les différentes valeurs que peut prendre chaque activité variable. Des constructions sont également proposées afin de spécifier comment connecter un fragment de processus à un processus sur lequel de la variabilité est définie. Mais cette approche est dépendante de BPMN, puisque les constructions proposées doivent être adaptées pour être utilisées avec un autre langage de modélisation de processus. Par exemple, il faut définir une correspondance entre les éléments de modèle BPMN et ceux de l'autre langage pour savoir comment réutiliser les constructions proposées par ABIS.

3.2.2.4 Configuration de processus par composition

L'approche proposée par Istoan [Ist13] permet de gérer la variabilité sur les aspects comportementaux d'un produit logiciel, alors que les approches actuelles se concentrent plutôt sur la gestion de la variabilité au niveau de la structure de produits logiciels. Le comportement d'un produit logiciel est défini à l'aide d'un processus métier. L'approche proposée permet de définir un ensemble de fragments de processus et de composer ces fragments pour dériver le processus correspondant à un produit logiciel spécifique. Le concept d'*interface* est introduit afin de spécifier les éléments d'un fragment de processus qui peuvent être connectés avec d'autres fragments de processus et comment ces éléments peuvent être connectés. Cependant, aucun moyen n'est

proposé afin de pouvoir définir des interfaces sans modifier le langage de modélisation de processus utilisé.

3.2.2.5 Association de plusieurs techniques

L'approche *Configurative Process Modeling* [BDKK04] permet de créer différentes vues d'un même modèle de processus afin de ne garder que les informations qui sont pertinentes pour un groupe d'utilisateurs. Mais certains des mécanismes proposés par cette approche peuvent également être utilisés afin de configurer des modèles de processus agrégats. Ainsi, cette approche permet d'associer des termes à des éléments d'un modèle afin de spécifier dans quels cas ceux-ci peuvent être sélectionnés. De plus, l'approche *Configurative Process Modeling* a été étendue afin de permettre la configuration de processus par agrégation (c'est-à-dire par composition de fragments de processus), par instanciation (c'est-à-dire affectation d'une valeur concrète à un concept abstrait) ou par spécialisation (ajout, suppression ou modification d'éléments de modèle) [BDK07]. Cependant, pour être mis en œuvre ces mécanismes requièrent d'étendre le langage de modélisation de processus utilisé, afin de pouvoir associer à des éléments de modèles des informations concernant leurs cas d'utilisation ou afin de pouvoir définir quels sont les éléments qui peuvent être instanciés, spécialisés ou composés avec d'autres éléments. Là encore, aucun moyen n'est proposé afin de réaliser cette extension sans modifier le langage de modélisation de processus utilisé.

3.2.3 Transformation en une structure pivot

Les approches de cette catégorie transforment les modèles de processus en des structures pivot sur lesquelles sont définis des mécanismes permettant de définir et de résoudre la variabilité. Une fois la variabilité résolue sur la structure pivot, une transformation de cette structure vers le métamodèle de processus utilisé permet d'obtenir un processus résolu. Les mécanismes nécessaires à la gestion de la variabilité ne sont donc plus couplés aux langages de modélisation de processus.

L'approche GV2BPMN (*Goal-Oriented Variability Analysis toBPMN*) [SCS10] propose de transformer les modèles de processus en des modèles de buts [YLL+08], afin de faciliter la sélection d'un processus d'une famille. Un modèle de buts permet en effet de capturer les différents objectifs qu'un système doit atteindre, ainsi que la variabilité entre ces objectifs. Cette approche permet de gérer la variabilité au niveau du séquencement des unités de travail d'une famille de processus.

L'approche de Meerkamm [Mee10] permet de gérer la variabilité au niveau des ressources nécessaires à la réalisation d'une unité de travail, en plus de la variabilité au niveau du séquencement de ces unités de travail. Elle permet de plus de différencier les variantes de processus (qui correspondent à des processus différents d'une même famille) des alternatives, c'est-à-dire des choix qui sont faits au moment de l'exécution d'un processus. Cette approche propose de transformer les modèles de processus en une structure arborescente qui permet de capturer à la fois les unités de travail et leurs ressources, et qui permet de gérer leur variabilité.

L'approche de Kumar et Yao [KY12] permet de gérer la variabilité au niveau du séquencement des unités de travail d'un processus et au niveau des ressources liées à ces unités de travail. Elle permet également de découpler la logique métier des processus afin d'éviter la modification des processus lorsque la logique métier change. De plus, elle propose des techniques pour faciliter la recherche et la récupération de variantes de processus stockées dans un dépôt. Afin de gérer la variabilité des processus tout en les découplant de la logique métier, cette approche consiste à appliquer des opérations de modification à un processus afin d'en dériver des variantes. La logique métier est donc capturée dans ces opérations. Ainsi, lorsque cette logique métier change, seules les opérations sont modifiées et pas les processus. Afin que les opérations de modifications puissent être réutilisées quel que soit langage de modélisation de processus, celles-ci sont définies sur une structure arborescente en laquelle sont transformés les modèles de processus.

Mais ces approches restent tout de même dépendantes des langages de modélisation de processus. En effet, elles nécessitent la définition d'une transformation pour chaque langage de modélisation de processus.

3.2.4 Spécification de la variabilité sans réutilisation automatique

Pour terminer, nous présentons des approches se concentrant sur la spécification de la variabilité d'une famille de processus, mais ne permettant pas la réutilisation automatique des processus de cette famille.

L'approche de Razavian et Khosravi [RK08] permet de modéliser la variabilité dans les processus métiers en utilisant UML. Pour ce faire, un profil UML est proposé afin de modéliser la variabilité dans des diagrammes d'activité UML. Ce profil propose plusieurs stéréotypes permettant de spécifier la variabilité au niveau des activités, des flots de contrôle, des données consommées et produites par les activités et des magasins de données (éléments utilisés pour persister les données). Deux types de variabilité peuvent être spécifiés : l'optionnalité et les alternatives. Cette approche ne propose cependant pas de mécanisme permettant de résoudre la variabilité et de dériver un processus en fonction de cette résolution de variabilité. De plus, aucun mécanisme n'est proposé afin de pouvoir appliquer les concepts proposés pour spécifier la variabilité sans modifier le langage de modélisation de processus utilisé.

L'approche de Simmonds et Bastarrica [SB11] investigue l'utilisation des BFM et des OVM (*Orthogonal Variability Model*) [PBL05] afin de spécifier la variabilité des processus. Les BFM sont indépendants du langage de modélisation de processus utilisé. Cependant, ils ne fournissent pas de mécanisme permettant de faire le lien avec les modèles de processus, ce qui empêche de pouvoir dériver un processus en fonction de la résolution de la variabilité spécifiée par le BFM. Les OVM fournissent quant à eux un mécanisme permettant de faire le lien avec un modèle de processus. Mais celui-ci implique de modifier le langage de modélisation de processus utilisé. En effet, afin de pouvoir définir de la variabilité sur les éléments de ce langage, ceux-ci doivent hériter de certains des éléments des OVM. D'autre part, les OVM ne fournissent pas de mécanisme pour dériver un processus.

Afin de faciliter la maintenance d'une famille de processus, l'approche de Derguech et Bhiri [DB10] consiste en l'utilisation d'une structure arborescente afin de définir une famille de processus et propose un support pour mettre à jour cette structure. Les nœuds de la structure arborescente sont des fragments de processus. Lorsqu'un nœud a des enfants, cela signifie que ce nœud est un point de variation et les enfants représentent les différentes variantes de ce point de variation. Cependant, l'aspect sélection et dérivation d'un processus de cette structure arborescente n'est pas traité par cette approche. De plus, aucun mécanisme n'est proposé afin de permettre l'utilisation de cette structure indépendamment du langage de modélisation de processus utilisé.

Les travaux de Simidchieva et al. [SCO07] caractérisent ce que devrait comprendre une famille de processus et introduisent une approche formelle pour définir une famille de processus en fonction de cette caractérisation. L'approche consiste à modéliser un processus de base ainsi que les éléments de processus nécessaires à la création d'autres variantes de ce processus. Une variante particulière de processus est obtenue en augmentant le processus de base à l'aide de ces éléments de processus, sélectionnés en fonction d'une spécification de buts de processus. La notion de spécification de buts de processus est proche de la notion d'exigences de projet. Cette approche est assez abstraite pour être indépendante du formalisme utilisé pour définir les processus. Cependant, elle ne fournit pas de mécanisme concret permettant d'obtenir le processus correspondant aux exigences d'un projet.

Fiorini et al. [FPLPdL01] proposent une architecture de réutilisation de processus, qui permet d'organiser et de décrire des processus et de les réutiliser. Cette architecture comprend un *framework* de processus, qui est ici un processus capturant les différents processus d'un domaine. Certaines parties de ce *framework* peuvent s'appuyer sur des processus standards (par exemple CMM [Ins95]) ou des patrons de processus capturés dans un dépôt. À ce *framework* sont associées des directives de réutilisation, qui définissent les activités du *framework* qui sont obligatoires, optionnelles, celles qui doivent être spécialisées, leur contenu... Une activité peut par exemple être spécialisée en utilisant un patron de processus du dépôt, récupéré à l'aide d'une recherche par mots clés. Mais malgré ces directives, la configuration du *framework* reste manuelle. Il est également possible de récupérer des patrons de processus pour les réutiliser indépendamment du *framework*. L'architecture est assez abstraite pour être utilisée avec des langages de modélisation de processus différents.

3.2.5 Synthèse sur l'ingénierie des lignes de processus

Les approches s'appuyant sur l'ingénierie des lignes de processus afin de réutiliser les processus de développement logiciel sont, à notre connaissance, dépendantes du langage de modélisation utilisé pour définir les processus de la ligne, comme résumé par le tableau 3.4. En effet, certaines utilisent les concepts natifs d'un tel langage pour gérer la variabilité d'une famille de processus. D'autres approches étendent un langage de modélisation de processus existant avec des concepts permettant de gérer la variabilité. Cependant, dans ces deux cas, il peut être nécessaire d'adapter ces concepts pour les utiliser avec un autre langage de modélisation de processus. De plus, la plu-

part de ces approches ne propose pas de mécanisme permettant d'utiliser ces concepts sans modifier un langage de modélisation de processus. Or, la modification d'un tel langage implique que les outils qui lui sont spécifiques (ex : modeleurs) deviennent inutilisables à moins d'être eux-aussi adaptés. D'autres approches résolvent partiellement le problème de la dépendance vis-à-vis du langage de modélisation de processus en transformant les modèles de processus en des structures pivot permettant de gérer la variabilité. Ces approches sont cependant également dépendantes du langage de modélisation de processus. En effet, une transformation doit être définie pour chaque langage. Enfin, quelques approches se concentrent uniquement sur l'aspect spécification de la variabilité, et certaines d'entre elles sont indépendantes des langages de modélisation de processus. Mais ces approches ne permettent pas la réutilisation automatique des processus dont la variabilité a été spécifiée.

Approche	Concepts proposés	Apports réutilisation processus	Limites réutilisation processus	Déf. des processus en intention	Indépendante du formalisme pour définir les processus
SPEM 2.0, d'après Alegría et al. [HABQO11] et Alegría et Bastarrica [AB12]	Utilisation de SPEM 2.0	Déf. d'un ensemble de processus et réutilisation des processus de cet ensemble	aucune	oui	non
KobrA [ABM00]	Extension d'un langage de modélisation de processus	Déf. d'un ensemble de processus et réutilisation des processus de cet ensemble	aucune	oui	non
Superimposed Variants [CA05]	Extension d'un langage de modélisation de processus	Déf. d'un ensemble de processus et réutilisation des processus de cet ensemble	aucune	oui	non
C-EPC [RvdA07]	Extension d'EPC	Déf. d'un ensemble de processus et réutilisation des processus de cet ensemble	aucune	oui	non
C-iEPC [RDH+08, LRDtHM11]	Extension d'EPC	Déf. d'un ensemble de processus et réutilisation des processus de cet ensemble	aucune	oui	non
aEPC [RMvdT09]	Extension d'EPC	Déf. d'un ensemble de processus et réutilisation des processus de cet ensemble	aucune	oui	non
Configurative Process Modeling [BDK07]	Extension d'un langage permettant de définir des méthodes d'ingénierie logicielle	Déf. d'un ensemble de processus et réutilisation des processus de cet ensemble	aucune	oui	non

Approche	Concepts proposés	Apports réutilisation processus	Limites réutilisation processus	Déf. des processus en intention	Indépendante du formalisme pour définir les processus
ADOM [RBSS09, RBSS10]	Extension d'un langage de modélisation de processus	Déf. d'un ensemble de processus et réutilisation des processus de cet ensemble	aucune	oui	non
Configurable Workflow [GvdAJVLR08]	Extension d'un langage de modélisation de *workflows*	Déf. d'un ensemble de processus et réutilisation des processus de cet ensemble	aucune	oui	non
Provop (*PROcess Variants by OPtions*) [HBR10]	Extension d'un langage de modélisation de processus	Déf. d'un ensemble de processus et réutilisation des processus de cet ensemble	aucune	oui	non
ADORE [MBF13]	Extension d'un langage de modélisation de processus	Déf. d'un ensemble de processus et réutilisation des processus de cet ensemble	aucune	oui	non
vSPEM [MRGP08]	Extension de SPEM 2.0	Déf. d'un ensemble de processus et réutilisation des processus de cet ensemble	aucune	oui	non
Kulkarni et Barat [KB10]	Extension de BPMN	Déf. d'un ensemble de processus et réutilisation des processus de cet ensemble	aucune	oui	non
Nguyen et al. [NCH11]	Extension de BPMN	Déf. d'un ensemble de processus et réutilisation des processus de cet ensemble	aucune	oui	non
PESOA [PSWW05]	profil UML	Déf. d'un ensemble de processus et réutilisation des processus de cet ensemble	aucune	oui	non

Approche	Concepts proposés	Apports réutilisation processus	Limites réutilisation processus	Déf. des processus en intention	Indépendante du formalisme pour définir les processus
BPFM [MHY08]	profil UML	Déf. d'un ensemble de processus et réutilisation des processus de cet ensemble	aucune	oui	non
Ripon et al. [RTM10]	profil UML	Déf. d'un ensemble de processus et réutilisation des processus de cet ensemble	aucune	oui	non
Ternité [Ter09]	Métamodèle	Déf. d'un ensemble de processus et réutilisation des processus de cet ensemble	aucune	oui	non
Ciuksys et Caplinskas [CC07]	Métamodèle	Déf. d'un ensemble de processus et réutilisation des processus de cet ensemble	aucune	oui	non
Acher et al. [ACG+12]	Extension des services qui composant un *workflow*	Déf. d'un ensemble de processus et réutilisation des processus de cet ensemble	aucune	oui	non
ABIS [WKK+11]	Extension de BPMN	Déf. d'un ensemble de processus et réutilisation des processus de cet ensemble	aucune	oui	non
Istoan [Ist13]	Extension d'un langage de modélisation de processus	Déf. d'un ensemble de processus et réutilisation des processus de cet ensemble	aucune	oui	non
Configurative Process Modeling [BDK07]	Extension d'un langage de modélisation de processus	Déf. d'un ensemble de processus et réutilisation des processus de cet ensemble	aucune	oui	non

Approche	Concepts proposés	Apports réutilisation processus	Limites réutilisation processus	Déf. des processus en intention	Indépendante du formalisme pour définir les processus
GV2BPMN [SCS10]	Transformation de modèles de processus en modèles de buts	Déf. d'un ensemble de processus et réutilisation des processus de cet ensemble	aucune	oui	non
Meerkamm [Mee10]	Transformation des modèles de processus en une structure arborescente	Déf. d'un ensemble de processus et réutilisation des processus de cet ensemble	aucune	oui	non
Kumar et Yao [KY12]	Transformation d'un processus en une structure arborescente	Déf. d'un ensemble de processus et réutilisation des processus de cet ensemble	aucune	oui	non
Razavian et Khosravi [RK08]	profil UML pour spécifier la variabilité dans les diagrammes d'activités	spécification de la variabilité des processus	résolution de la variabilité, dérivation de processus	oui	non
Simmonds et Bastarrica [SB11]	BFM et OVM pour spécifier la variabilité des processus	spécification de la variabilité des processus	dérivation de processus	oui	oui BFM, non OVM

Approche	Concepts proposés	Apports réutilisation processus	Limites réutilisation processus	Déf. des processus en intention	Indépendante du formalisme pour définir les processus
Derguech et Bhiri [DB10]	Structure arborescente pour définir une famille de processus	spécification de la variabilité des processus	résolution de la variabilité, dérivation de processus	oui	non
Simidchieva et al. [SCO07]	Caractérisation du contenu d'une famille de processus et approche pour définir une telle famille	spécification de la variabilité des processus	dérivation de processus	oui	oui
Fiorini et al. [FPLPdL01]	Architecture de réutilisation de processus	spécification de la variabilité des processus	dérivation de processus	oui	oui

TABLE 3.4 – Évaluation des approches s'appuyant sur l'ingénierie des lignes de processus

3.3 Synthèse

L'approche de Sadovykh et Abherve permet de savoir à quels moments d'un projet réutiliser des automatisations de TMR. Cette approche ne permet cependant pas de savoir pour quels projets réutiliser des automatisations de TMR. Une solution consiste donc à réutiliser les processus de développement logiciel afin de réutiliser les automatisations de TMR qui leurs sont liées. Parmi les approches supportant la réutilisation des processus de développement logiciel, certaines ne sont pas suffisantes pour mettre en œuvre cette réutilisation. D'autres approches permettent effectivement de réutiliser les processus de développement logiciel. Cependant, soit celles-ci ne proposent pas de mécanisme pour définir les processus en intention, soit elles sont dépendantes du langage de modélisation de processus utilisé. Le tableau 3.5 résume dans quelle catégorie se trouve chaque approche supportant la réutilisation des processus. En conclusion, aucune des approches existantes ne permet de définir un ensemble de processus en intention et de dériver un processus de cet ensemble, tout en étant indépendante du formalisme utilisé pour définir les processus. De plus, aucune approche ne traite la problématique de l'automatisation des processus de développement logiciel réutilisés, ce qui est loin d'être trivial. En effet, les contraintes d'un projet peuvent imposer de commencer l'exécution d'un processus alors que toutes les exigences de ce projet ne sont pas encore déterminées. Par exemple, afin de gagner du temps, il est possible de commencer l'exécution d'un processus de développement d'une application web Java (par exemple en écrivant les spécifications fonctionnelles) alors que le *framework* pour l'IHM de l'application à développer (par exemple Struts, JSF, Flex ou GWT) n'est pas encore choisi. Mais comment exécuter un processus s'il n'est que partiellement défini à cause des exigences non déterminées ?

Approche	Suffisante[1] pour la réutilisation de processus	Définition des processus en intention	Indépendante du formalisme pour définir les processus
Promote [CDCC+13]	non	non	oui
Pérez et al. [PEM95]	non	non	oui
Alexander et Davis [AD91]	non	non	oui
Sharon et al. [SdSSB+10]	non	non	oui
Lu et Sadiq [LS07]	oui	non	non
Song et Osterweil [SO98]	oui	non	oui
Intelligent Workflow Designer [IMCH+07]	non	non	non
Design by Selection [ASKW11]	non	non	non
CAT [KSG04]	non	non	non

Approche	Suffisante [1] pour la réutilisation de processus	Définition des processus en intention	Indépendante du formalisme pour définir les processus
Aldin [Ald10]	non	non	oui
SPEM 2.0, d'après Alegría et al. [HABQO11] et Alegría et Bastarrica [AB12]	oui	oui	non
KobrA [ABM00]	oui	oui	non
C-EPC [RvdA07]	oui	oui	non
C-iEPC [RDH+08, LRDtHM11]	oui	oui	non
aEPC [RMvdT09]	oui	oui	non
Configurative Process Modeling [BDK07]	oui	oui	non
Superimposed Variants [CA05]	oui	oui	non
ADOM [RBSS09, RBSS10]	oui	oui	non
Configurable Workflow [GvdAJVLR08]	oui	oui	non
Provop (*PROcess Variants by OPtions*) [HBR10]	oui	oui	non
vSPEM [MRGP08]	oui	oui	non
Kulkarni et Barat [KB10]	oui	oui	non
Nguyen et al. [NCH11]	oui	oui	non
PESOA [PSWW05]	oui	oui	non
BPFM [MHY08]	oui	oui	non
Ripon et al. [RTM10]	oui	oui	non
Ternité [Ter09]	oui	oui	non
Ciuksys et Caplinskas [CC07]	oui	oui	non
Acher et al. [ACG+12]	oui	oui	non
ABIS [WKK+11]	oui	oui	non
Configurative Process Modeling [BDK07]	oui	oui	non
Istoan [Ist13]	oui	oui	non

Approche	Suffisante[1] pour la réutilisation de processus	Définition des processus en intention	Indépendante du formalisme pour définir les processus
ADORE [MBF13]	oui	oui	non
GV2BPMN [SCS10]	oui	oui	non
Meerkamm [Mee10]	oui	oui	non
Kumar et Yao [KY12]	oui	oui	non
Razavian et Khosravi [RK08]	non	oui	non
Simmonds et Bastarrica [SB11]	non	oui	oui BFM, non OVM
Derguech et Bhiri [DB10]	non	oui	non
Simidchieva et al. [SCO07]	non	oui	oui
Fiorini et al. [FPLPdL01]	non	oui	oui

TABLE 3.5 – Évaluation de l'ensemble des approches supportant la réutilisation des processus

Concernant la capacité des automatisations de TMR à être réutilisées à travers leurs différents cas d'utilisation, certaines approches supportent le développement de composants logiciels réutilisables. Par exemple, l'ingénierie des lignes de produits logiciels [PBL05] s'appuie sur la spécification de la variabilité de produits logiciels afin de produire des artéfacts de développement réutilisables. Plusieurs approches permettent d'implémenter des artéfacts réutilisables, comme par exemple la programmation orientée objet [Cox85], la programmation orientée aspect [CB05] ou encore l'ingénierie logicielle basée sur les composants [Crn02]. Mais pour être appliquées ces approches requièrent toutes au préalable d'avoir identifié la variabilité des artéfacts de développement, ce qu'elles ne permettent pas de faire.

Des approches permettent quant à elle d'identifier la variabilité de composants logiciels. FORM (*Feature-Oriented Reuse Method*) [KKL+98] est une méthodologie mettant en œuvre l'ingénierie des lignes de produits logiciels. À ce titre, elle supporte la conception et le développement d'architectures et de composants réutilisables, et le développement de produits logiciels à partir de ces artéfacts. FORM fournit des directives permettant d'identifier la variabilité des artéfacts de développement qui réalisent des produits logiciels, en fonction de la spécification de la variabilité de ces produits logiciels. Un BFM est utilisé pour spécifier la variabilité d'un produit logiciel et l'identification de la variabilité d'un composant logiciel est basée sur l'analyse de la hiérarchie du BFM. Un exemple de directive proposée par FORM est que la hiérarchie des composants qui réalisent un produit logiciel correspond en grande partie à la

1. Comme défini page 36, une approche est suffisante pour la réutilisation de processus si elle permet de définir un ensemble de processus et de dériver un processus de cet ensemble.

hiérarchie du BFM.

L'approche de Lee et Kang [LK04] étend l'approche FORM et s'appuie sur l'analyse des dépendances opérationnelles entre les caractéristiques capturées par le BFM (c'est-à-dire lorsque la réalisation d'une caractéristique dépend de la réalisation d'autres caractéristiques) afin d'identifier la variabilité d'un composant logiciel. Un exemple de directive proposée par cette approche est que des caractéristiques alternatives peuvent implémenter une interface commune.

L'approche de Lee et al. [LKCC00] étend elle aussi l'approche FORM afin d'identifier des objets réutilisables pour l'implémentation de composants à partir de l'analyse d'un BFM. Un exemple de directive est que des catégories d'objets peuvent être déduites de catégories de caractéristiques.

Dans le cas de l'application de ces approches à des automatisations de TMR qui réalisent des processus de développement logiciel, le BFM spécifierait la variabilité d'une famille de processus et les automatisations de TMR seraient les composants logiciels dont la variabilité doit être identifiée. Cependant, il ne serait pas possible de réutiliser les directives proposées par ces approches telles qu'elles sont. En effet, chaque caractéristique du BFM (correspondant dans ce cas à un fragment de processus) n'est pas systématiquement réalisée par une automatisation de TMR. Ces approches ne sont donc pas adaptées pour identifier la variabilité d'automatisations de TMR.

Une autre méthode permet d'identifier les parties réutilisables d'un composant en identifiant les parties communes entre les variantes des points de variation de ce composant [ZXD05]. Mais cette méthode requière au préalable d'avoir identifié les points de variation et les variantes d'un composant. Cette méthode permet donc d'aller plus loin dans l'identification de la variabilité d'un composant, mais ne peut pas être appliquée si des parties variables et communes d'un composant n'ont pas déjà été identifiées.

En conclusion, il manque une approche permettant de réutiliser les processus de développement logiciel qui soit à la fois indépendante du langage de modélisation de processus utilisé, qui offre les mécanismes nécessaires à la définition en intention d'un ensemble de processus, et qui intègre réutilisation et automatisation des processus. Il manque également une approche permettant d'identifier la variabilité d'automatisations de TMR.

Deuxième partie

Automatisation des tâches manuelles récurrentes pilotée par les processus de développement logiciel

Les travaux présentés dans cette partie ont fait l'objet d'une publication :

Emmanuelle Rouillé, Benoît Combemale, Olivier Barais, David Touzet and Jean-Marc Jézéquel. Integrating Software Process Reuse and Automation. In Proceedings of the 20th Asia-Pacific Software Engineering Conference, APSEC '13, 2013 [RCB+13b].

Nous présentons dans cette partie la contribution principale de cette thèse, à savoir notre approche pilotant la réutilisation d'automatisations de TMR (Tâches Manuelles Récurrentes) par les processus de développement logiciel. La figure 3.4 illustre le principe général de cette approche.

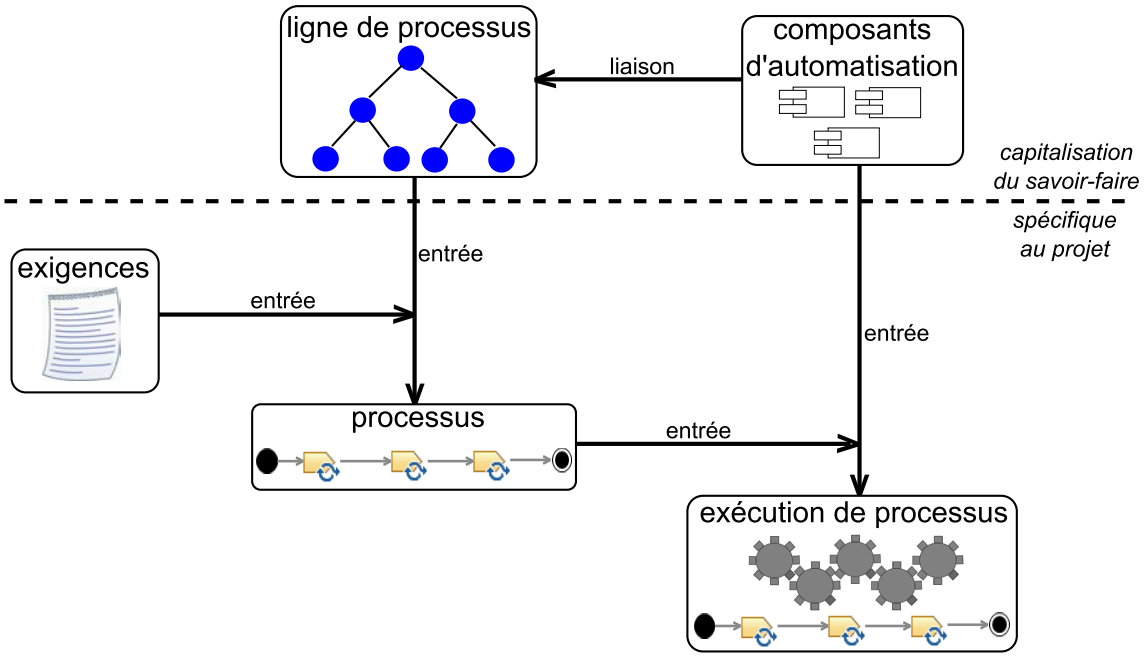


FIGURE 3.4 – Principe général de la contribution principale

La réutilisation des automatisations de TMR consiste dans notre approche à réutiliser les processus de développement logiciel auxquels ces automatisations sont liées. Notre approche s'appuie sur l'ingénierie des lignes de processus afin de réutiliser les processus de développement logiciel. Une ligne de processus est définie en intention et un processus peut être réutilisé en le dérivant de cette ligne, en fonction des exigences des projets. L'ingénierie des lignes de processus permet non seulement de réutiliser les processus de développement logiciel, mais également tous les fragments de processus communs à plusieurs processus, puisque ceux-ci sont factorisés (définition en intention). De plus, la définition en intention d'une famille de processus permet d'en faciliter

la maintenance, puisque les fragments de processus communs à plusieurs processus ne sont mis à jour qu'une seule fois en cas d'évolution. D'autres part, comme des langages de modélisation de processus de développement logiciel différents sont utilisés en fonction des exigences des utilisateurs [Zam01], nous proposons, dans le chapitre 4, une approche permettant de gérer la variabilité dans les processus de développement logiciel qui est indépendante du langage de modélisation de processus utilisé.

Les TMR à automatiser étant des tâches réalisées par ordinateur, nous utilisons des composants logiciels afin de les automatiser. Dans la suite de cette thèse, nous appelons *composants d'automatisation* (CA) un composant logiciel automatisant une TMR. Ces CA sont reliés aux unités de travail (tâches, activités...) des processus de la ligne de processus qu'ils automatisent. Cependant, un CA peut être lié à des unités de travail différentes, appartenant ou non à un même processus, ce qui peut induire de la variabilité au niveau de ce CA. Par exemple, un CA qui met du code source sous contrôle de version peut être utilisé lors de projets différents, pour lesquels le dépôt distant sur lequel partager le code source n'est pas le même. Nous proposons donc, dans le chapitre 5, une méthodologie fournissant un support à la création de CA qui soient réutilisables pour toutes les unités de travail auxquelles ils sont liés.

Un processus dérivé de la ligne de processus est automatisé en lançant au fur et à mesure de son exécution les CA qui lui sont liés. Le lien entre ces CA et les unités de travail du processus qu'ils automatisent permet de savoir à quel moment de l'exécution du processus exécuter les différents CA.

Chapitre 4

Gestion de la variabilité dans les processus

Nous présentons dans ce chapitre notre approche permettant de gérer la variabilité dans les processus de développement logiciel indépendamment du langage utilisé pour les définir. Nous commençons par introduire dans la section 4.1 l'exemple dont nous nous servons pour illustrer cette approche. La section 4.2 détaille ensuite l'approche en elle-même. Nous discutons cette approche dans la section 4.3 et nous en faisons la synthèse dans la section 4.4.

> Les travaux présentés dans ce chapitre ont fait l'objet d'une publication :
>
> Emmanuelle Rouillé, Benoît Combemale, Olivier Barais, David Touzet and Jean-Marc Jézéquel. Leveraging CVL to Manage Variability in Software Process Lines. In Proceedings of the 19th Asia-Pacific Software Engineering Conference, APSEC '12, pages 148-157, 2012 [RCB+12].

4.1 Exemple illustratif : une famille de processus de métamodélisation

Nous introduisons dans cette section un exemple de famille de processus, que nous utilisons ensuite pour illustrer l'approche proposée dans ce chapitre. Il s'agit d'une famille simplifiée de processus de métamodélisation, issue de l'équipe de recherche Triskell [1], au sein de laquelle cette thèse a été réalisée. Un processus de métamodélisation consiste à définir et outiller un langage de modélisation spécifique à l'expression d'une préoccupation particulière, en tirant parti pour ce faire des concepts et outils

1. http://triskell.irisa.fr/

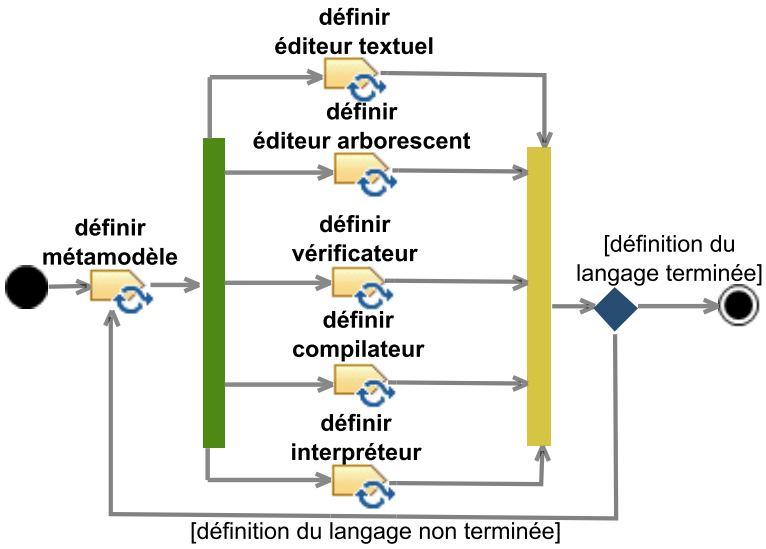


FIGURE 4.1 – Flot de tâches de l'exemple illustratif de processus de métamodélisation

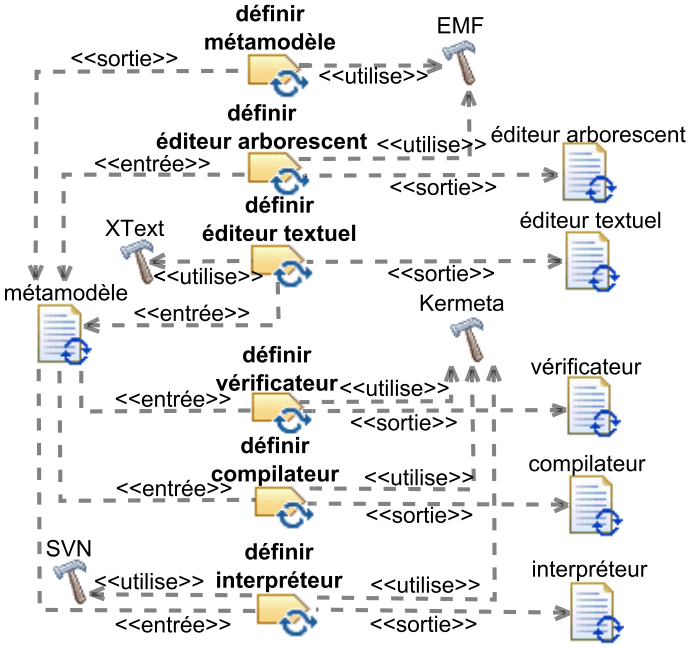


FIGURE 4.2 – Ressources de l'exemple illustratif de processus de métamodélisation

du domaine de l'IDM. Un tel langage permet de mettre en œuvre la séparation des préoccupations et ainsi maîtriser la complexité des systèmes logiciels [JCV12]. Nous introduisons d'abord un exemple simplifié de processus de métamodélisation, illustré par les figures 4.1 et 4.2. Plus précisément, la figure 4.1 illustre le flot de tâches de ce processus, tandis que la figure 4.2 illustre les ressources (outils, entrées et sorties) qui

entrent en jeu lors de la réalisation de ces tâches. Nous détaillons ensuite des exemples de variantes de ce processus.

L'exemple de processus de métamodélisation présenté ici est un processus itératif qui commence avec la définition d'un métamodèle. Des tâches parallèles suivent la définition de ce métamodèle : définition d'un éditeur textuel et d'un éditeur arborescent permettant la création de modèles conformes au métamodèle, définition d'un vérificateur permettant de vérifier qu'un modèle est bien conforme à son métamodèle, définition d'un interpréteur et définition d'un compilateur. L'outil utilisé pour définir le métamodèle et l'éditeur arborescent est EMF [2]. L'outil XText [3] est utilisé pour définir l'éditeur textuel. L'outil utilisé pour définir le vérificateur, l'interpréteur et le compilateur est Kermeta [4], un environnement de métamodélisation. L'interpréteur est mis sous contrôle de version en utilisant SVN.

Il existe des variantes de ce processus de métamodélisation. Par exemple, en fonction des exigences des projets, les tâches de définition d'un éditeur textuel ou arborescent, d'un interpréteur, d'un compilateur et d'un vérificateur peuvent ne pas avoir lieu. D'autre part, toujours selon les exigences des projets, l'outil EMFText [5] peut être utilisé à la place de XText et Git peut remplacer SVN.

4.2 Approche

Des langages de modélisation de processus différents peuvent être utilisés en fonction des exigences de leurs utilisateurs. Il est donc intéressant, à des fins de réutilisation, d'avoir une approche permettant de gérer la variabilité dans les processus qui soit indépendante de ces langages. Par *indépendante* nous entendons qu'une approche n'ait pas à être adaptée pour être utilisée avec des langages de modélisation de processus différents et qu'elle ne nécessite pas non plus de modifier ces langages. Cela permet de pouvoir réutiliser à la fois l'approche et les langages de modélisation de processus, ainsi que les outils associés. Nous avons vu en section 2.3.2 que CVL était un langage permettant de spécifier et de résoudre de la variabilité sur des modèles, quel que soit leur métamodèle pour peu qu'il se conforme au MOF. Nous utilisons donc ici CVL afin de gérer la variabilité dans les processus de développement logiciel indépendamment de leur langage de modélisation. Nous présentons dans ce chapitre notre approche, appelée CVL4SP (*CVL for Software Processes*). Il s'agit de la première sous-contribution de cette thèse. La figure 4.3 illustre la partie de la contribution principale que nous détaillons ici.

2. http://www.eclipse.org/modeling/emf/
3. http://www.eclipse.org/Xtext/
4. http://www.kermeta.org/
5. http://www.emftext.org

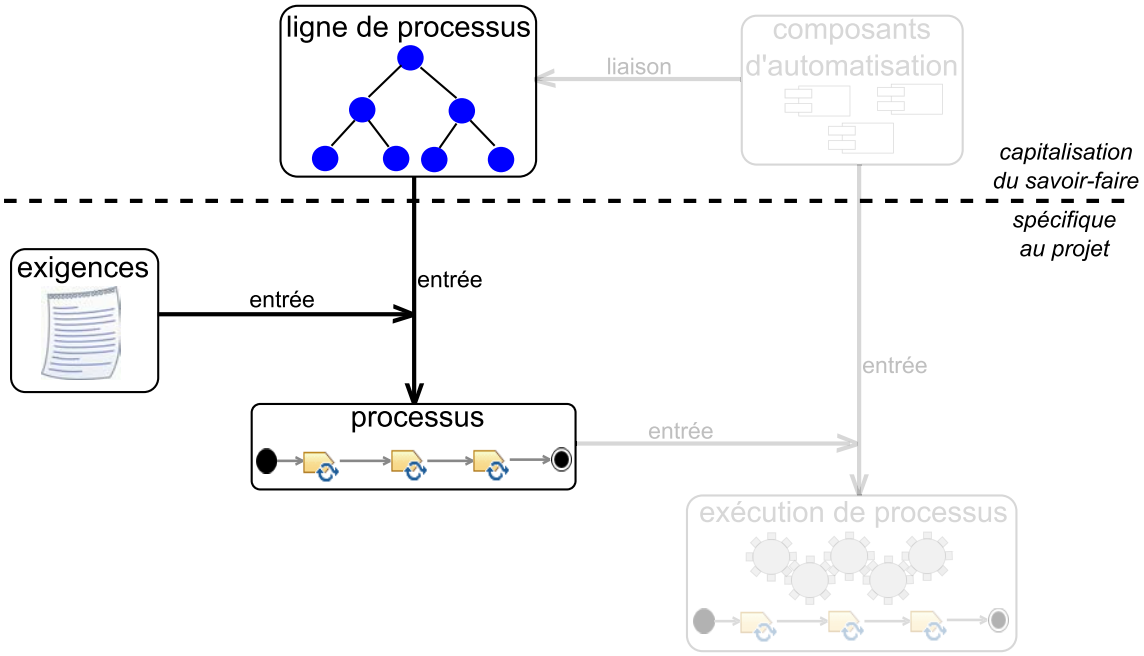


FIGURE 4.3 – Partie de la contribution principale réalisée par CVL4SP

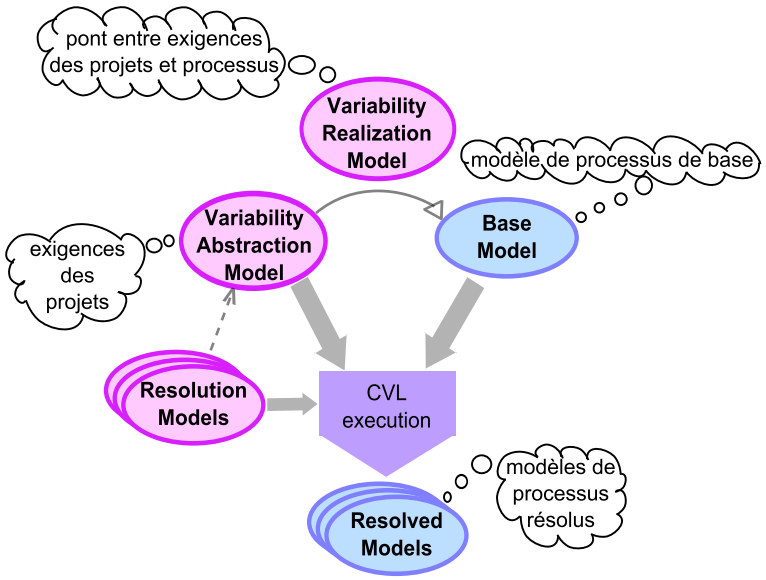


FIGURE 4.4 – Apperçu de l'approche consistant à utiliser CVL pour gérer la variabilité dans les processus de développement logiciel

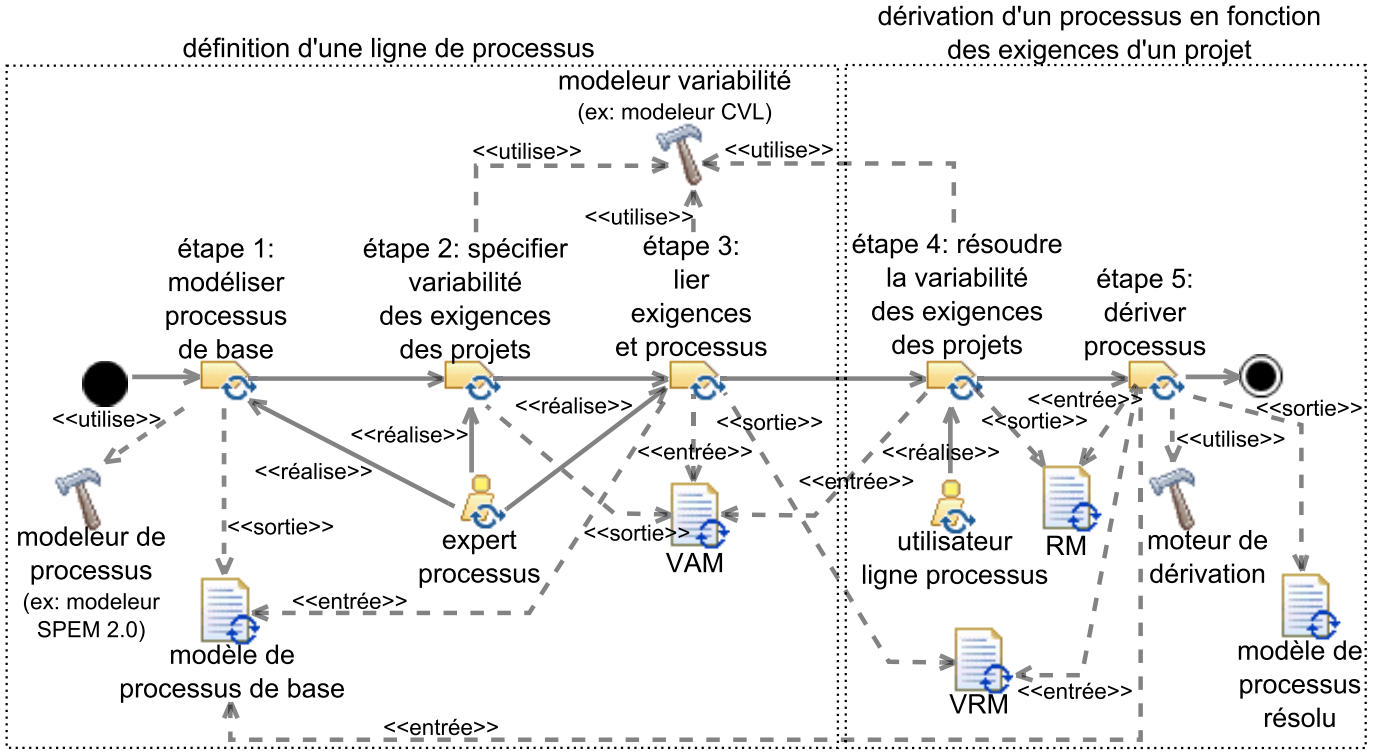


FIGURE 4.5 – Processus de l'approche consistant à utiliser CVL pour gérer la variabilité dans les processus de développement logiciel

Comme illustré par la figure 4.4, le principe général de CVL4SP consiste à utiliser le VAM (*Variability Abstraction Model*, cf. section 2.3.2) de CVL afin de capturer la variabilité des exigences des projets. Nous utilisons le modèle de base de CVL afin de capturer les éléments de processus nécessaires pour définir la famille de processus attendue. Ce modèle de base devient alors un modèle de processus de base. Grâce au VRM (*Variability Realization Model*, cf. section 2.3.2) de CVL, nous faisons le lien entre la variabilité des exigences des projets capturée dans le VAM et les éléments de processus du modèle de processus de base. Plus précisément, nous utilisons le VRM afin de définir comment dériver un processus du modèle de processus de base en fonction des exigences des projets. Nous utilisons le RM (*Resolution Model*, cf. section 2.3.2) afin de sélectionner les exigences pour un projet donné. Enfin, le modèle résolu de CVL contient le processus dérivé du modèle de processus de base en fonction des exigences sélectionnées dans le RM. On parle alors de modèle de processus résolu. Notre approche préserve la séparation entre les exigences des projets et les processus, puisque les aspects exigences et processus sont définis dans des modèles distincts (le VAM et le modèle de processus de base), qui ne sont pas liés directement l'un à l'autre mais par l'intermédiaire du VRM. Cela permet donc de pouvoir réutiliser et faire évoluer les aspects exigences et processus indépendamment l'un de l'autre. De plus, en capturant dans le VAM la variabilité des exigences des projets et non la variabilité des processus, notre approche permet de relier directement les exigences des projets aux processus, en évitant une couche intermédiaire de définition de la variabilité des processus.

La figure 4.5 présente le processus de notre approche. Elle implique deux rôles : un expert processus, qui connaît les différents processus d'une entreprise ainsi que leurs contextes d'utilisation, et un utilisateur d'une ligne de processus, qui est impliqué dans un projet et a besoin d'un processus spécifique à ce projet. L'expert processus capture la variabilité des exigences des projets et fait le lien entre ces exigences et une ligne de processus (étapes 1 à 3). Ensuite, l'utilisateur de cette ligne de processus lance la dérivation automatique d'un processus (de cette ligne) en fonction des exigences d'un projet spécifique (étapes 4 et 5). Les étapes 4 et 5 ont lieu à chaque fois qu'un utilisateur d'une ligne de processus souhaite dériver un processus. L'expert processus et l'utilisateur d'une ligne de processus utilisent l'outillage associé à CVL pour réaliser les étapes 2 à 5. N'importe quel langage de modélisation de processus de développement logiciel peut être utilisé pour réaliser l'étape 1.

Nous détaillons dans la suite de cette section les différentes étapes de notre approche. Nous les illustrons à l'aide de l'exemple illustratif introduit dans la section 4.1.

4.2.1 Définition de la ligne de processus de développement logiciel

4.2.1.1 Méthodologie pour la modélisation des éléments de processus (étape 1)

Approche L'expert processus modélise ici les éléments de processus requis pour définir la famille de processus attendue.

Dans un premier temps, l'expert processus modélise, dans un modèle de processus

de base, le processus de l'entreprise le plus souvent utilisé. Nous ne traitons pas dans cette thèse de la manière de déterminer le processus le plus souvent utilisé. Ensuite, l'expert processus modélise tous les autres éléments de processus qui n'appartiennent pas au processus le plus souvent utilisé et qui sont requis pour créer des variantes de ce processus. Dans la suite, nous appellerons ces éléments de processus des *éléments de processus externes*. Ces éléments de processus externes sont ajoutés au modèle contenant le processus le plus souvent utilisé (c'est-à-dire le modèle de processus de base), mais sans les relier à ce processus. Même si plusieurs processus utilisent un même élément de processus externe, celui-ci n'est modélisé qu'une seule fois dans le modèle de processus de base, afin de respecter la définition en intention de processus. De ce fait, quand un élément de processus commun à plusieurs processus évolue, il ne faut le mettre à jour qu'une seule fois. Quand différents processus utilisent un même élément de processus externe, ils peuvent requérir différentes valeurs pour les propriétés de cet élément de processus externe. Dans ce cas, dans le modèle de processus de base, l'expert processus affecte aux propriétés de cet élément de processus les valeurs correspondant au processus le plus utilisé parmi les processus qui utilisent cet élément de processus externe.

Nous détaillons maintenant comment notre méthodologie s'applique dans le cas de la modélisation avec SPEM 2.0. L'expert processus commence par modéliser les éléments de contenu de méthode qui serviront à définir le processus le plus souvent utilisé ainsi que les éléments de processus externes. L'expert processus modélise ensuite le processus le plus souvent utilisé ainsi que les éléments de processus externes dans différentes activités représentant des processus. De cette manière, il n'y a pas d'opération à effectuer sur le modèle de processus de base pour dériver le processus le plus souvent utilisé. En effet, en SPEM 2.0, un processus décrivant le cycle de vie complet d'un projet est décrit dans un type d'activité particulier, le processus. De cette manière, le processus le plus souvent utilisé est déjà décrit dans ce type d'activité.

Il est possible de modéliser le processus de base différemment. Par exemple, toutes les variantes de processus peuvent être modélisées dans le même processus en utilisant des branchements conditionnels. Il est également possible de modéliser des éléments de processus sans modéliser les relations entre eux et de les composer au moment de la dérivation. Une autre possibilité est de modéliser le processus commun à tous les processus d'une famille ainsi que les éléments de processus qui varient, sans les relier au processus commun. Par rapport à ces autres approches, notre méthodologie permet à un humain de visualiser directement au moins un processus d'une famille, qui plus est le processus le plus souvent utilisé de cette famille.

Illustration Nous détaillons maintenant le modèle de processus de base dans le cas de l'exemple illustratif de la famille de processus de métamodélisation. La figure 4.6 illustre les éléments de contenu de méthode de la famille de processus de métamodélisation. La définition de tâche *définir métamodèle* produit en sortie la définition de produit de travail *métamodèle* et requiert l'utilisation de l'outil *EMF*. Les définitions de tâche *définir éditeur arborescent*, *définir éditeur textuel*, *définir vérificateur*, *définir compilateur* et *définir interpréteur* prennent en entrée la définition de produit de travail *métamodèle*, et

produisent respectivement en sortie les définitions de produit de travail *éditeur arborescent*, *éditeur textuel*, *vérificateur*, *compilateur* et *interpréteur*. Comme pour les éléments de processus externes, on ne modélise que les propriétés correspondant au processus le plus utilisé qui utilise ces éléments de contenu de méthode. Ainsi, la définition de tâche *définir éditeur textuel* utilise la définition d'outil *XText* et non pas la définition d'outil *EMFText*, et la définition de tâche *définir interpréteur* utilise la définition d'outil *SVN* et non pas *Git*.

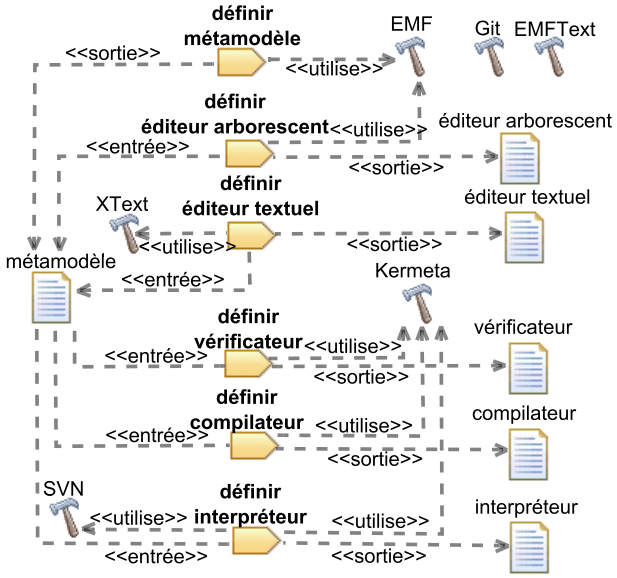


FIGURE 4.6 – Eléments de contenu de méthode de l'exemple illustratif

Comme illustré par la figure 4.7, le processus le plus utilisé de la famille de processus de métamodélisation consiste uniquement en la définition d'un métamodèle et d'un éditeur arborescent.

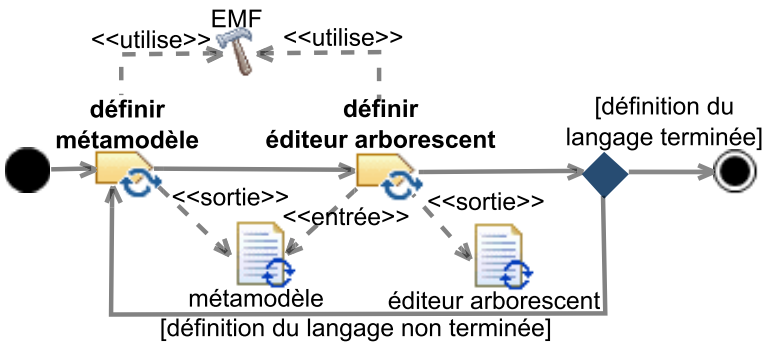


FIGURE 4.7 – Processus le plus souvent utilisé de l'exemple illustratif

La figure 4.8 illustre les fragments de processus représentant les éléments de processus externes de la famille de processus de métamodélisation. Ces éléments de processus

externes sont les tâches *définir éditeur textuel*, *définir vérificateur*, *définir interpréteur* et *définir compilateur*, avec leurs produits de travail respectifs (*éditeur textuel*, *vérificateur*, *interpréteur* et *compilateur*). Les éléments de processus externes comprennent également un nœud de jointure et un nœud de parallélisation, ainsi que les flots de contrôle nécessaires à la dérivation de processus de métamodélisation autres que celui qui est le plus utilisé.

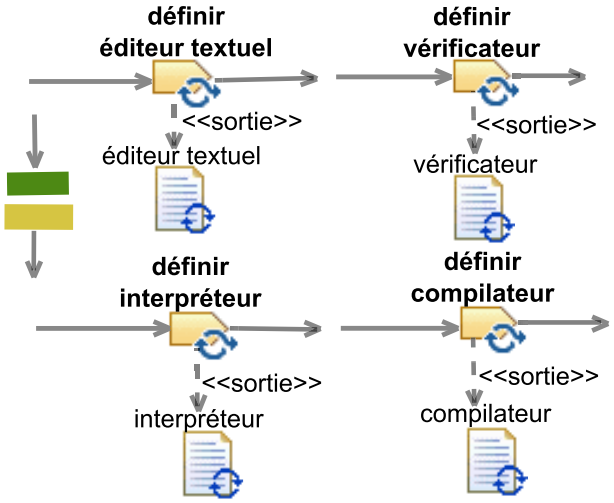


FIGURE 4.8 – Eléments de processus externes de l'exemple illustratif

4.2.1.2 Spécification de la variabilité des exigences des projets (étape 2)

Approche Dans la seconde partie de notre approche, l'expert processus s'appuie sur le VAM de CVL afin de spécifier la variabilité des exigences des projets.

Illustration La figure 4.9 présente un extrait du VAM des exigences des différents projets de métamodélisation de l'exemple illustratif. On y voit que les définitions d'un interpréteur, d'un compilateur, d'un vérificateur, d'un éditeur arborescent ainsi que d'un éditeur textuel (respectivement représentées par les choix *interpréteur*, *compilateur*, *vérificateur*, *éditeur arborescent* et *éditeur textuel*) sont des tâches optionnelles. La définition d'un éditeur textuel peut être réalisée soit avec l'outil XText soit avec l'outil EMFText (respectivement représentés par les choix *XText* et *EMFText*). L'utilisation d'un SCV (Système de Contrôle de Version), représenté par le choix *SCV*, est obligatoire, et il s'agit soit de SVN, soit de Git (respectivement représentés par les choix *SVN* et *Git*). Ce VAM contient également des choix dont la résolution est dérivée (*éditeur textuel parallèle*, *vérificateur parallèle*, *compilateur parallèle*, *interpréteur parallèle*, *éditeur arborescent parallèle* et *parallélisation*). Le choix *parallélisation* est automatiquement sélectionné lorqu'au moins deux tâches optionnelles de la famille de processus de métamodélisation sont sélectionnées. Chacun des choix *éditeur textuel parallèle*, *vérificateur parallèle*, *compilateur parallèle*, *interpréteur parallèle* et *éditeur arborescent parallèle* est

automatiquement sélectionné si le choix *parallélisation* est sélectionné et si le choix de nom correspondant (c'est-à-dire *éditeur textuel* pour *éditeur textuel parallèle*, *vérificateur* pour *vérificateur parallèle*, ...) est également sélectionné. Ces choix dont la résolution est dérivée ne reflètent pas la variabilité au niveau des exigences des projets. Nous allons voir par la suite qu'ils sont utiles pour savoir quelles opérations effectuer sur le modèle de processus de base afin de dériver un processus.

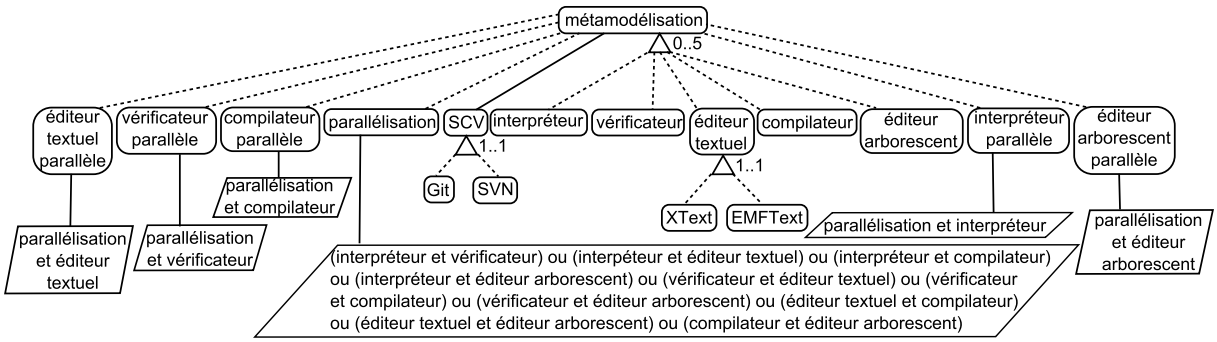


FIGURE 4.9 – Extrait du VAM de l'exemple illustratif

4.2.1.3 Liaison entre les exigences des projets et les processus (étape 3)

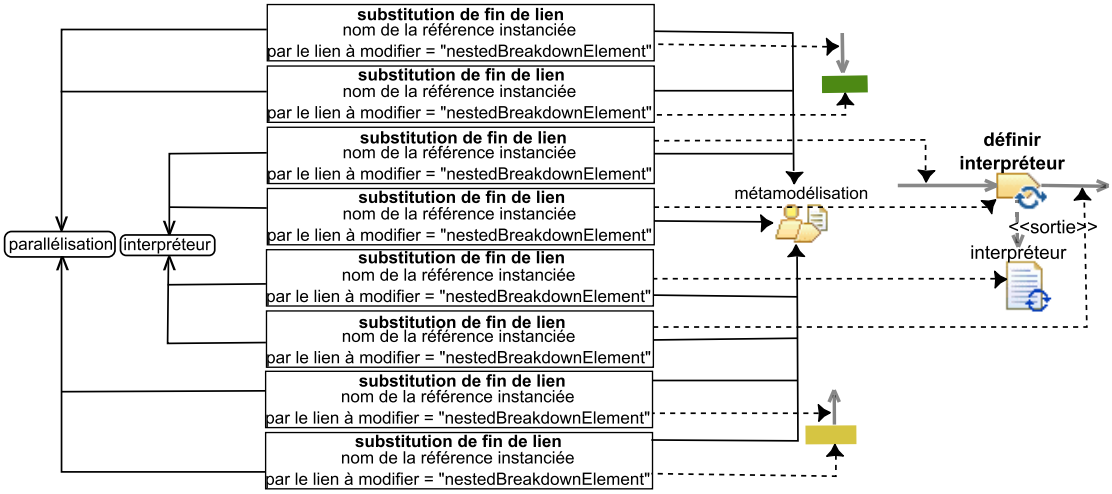
Approche Dans la troisième étape de notre approche, l'expert processus définit dans un VRM la liaison entre la variabilité des exigences des projets (définie à l'étape 2 dans le VAM) et les processus (définis à l'étape 1 dans le modèle de processus de base). Le VRM spécifie comment dériver un processus du modèle de processus de base en fonction des exigences des projets.

Illustration La figure 4.10 représente un extrait du VRM de l'exemple illustratif suffisant pour dériver un processus de métamodélisation avec définition d'un éditeur arborescent et d'un interpréteur. Pour des raisons de lisibilité, nous avons scindé ce VRM en deux dans les sous-figures 4.10(a) et 4.10(b).

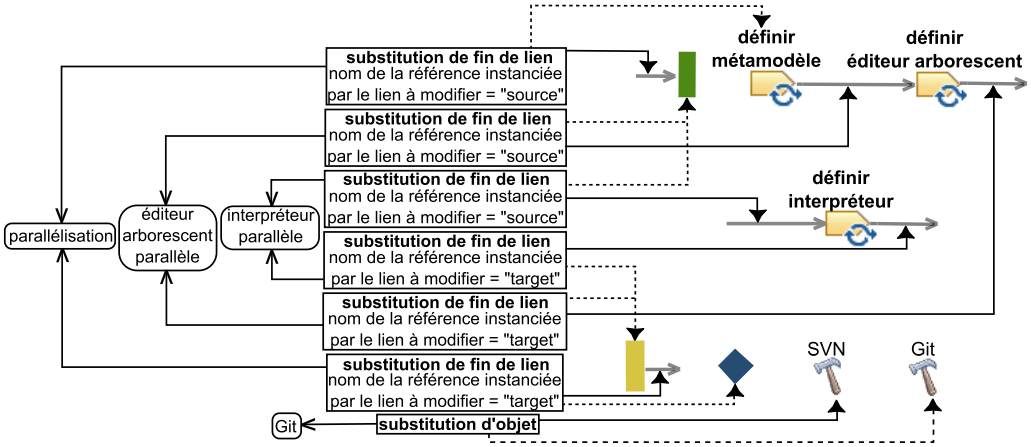
Dans la sous-figure 4.10(a), l'activité *métamodélisation* représente le processus le plus souvent utilisé. La première série d'opérations consiste à mettre dans ce processus les éléments de processus externes dont on a besoin pour dériver un processus en fonction des exigences. Ainsi, si le choix *interpréteur* est sélectionné, alors il faut mettre la tâche *définir interpréteur*, ses flots de contrôle entrants et sortants, ainsi que le produit de travail *interpréteur* dans l'activité *métamodélisation*. Si le choix *parallélisation* est sélectionné, cela signifie qu'au moins deux tâches optionnelles auront été sélectionnées et donc que le processus dérivé contiendra de la parallélisation car les tâches optionnelles sont exécutées en parallèle dans notre exemple. Il faut donc, dans ce cas, mettre le nœud de parallélisation, son flot de contrôle entrant, le nœud de jointure et son flot de contrôle sortant dans l'activité *métamodélisation*.

Détaillons maintenant la sous-figure 4.10(b). Si le choix *parallélisation* est sélectionné,

alors la source du flot de contrôle entrant du nœud de parallélisation doit être affectée à la tâche *définir métamodèle*. Le flot de contrôle sortant du nœud de jointure doit de plus être affecté au nœud de décision. Si le choix *éditeur arborescent parallèle* est sélectionné, cela signifie que le processus dérivé contiendra des tâches parallèles, et que la tâche *définir éditeur arborescent* en fera partie. Il faut donc, dans ce cas, affecter la source du flot de contrôle entrant de la tâche *définir éditeur arborescent* au nœud de parallélisation et il faut affecter la cible du flot de contrôle sortant de la tâche *définir éditeur arborescent* au nœud de jointure. De manière similaire, si le choix *interpréteur parallèle* est sélectionné, alors il faut affecter la source du flot de contrôle entrant de la tâche *définir interpréteur* au nœud de parallélisation et il faut affecter la cible du flot de contrôle sortant de la tâche *définir interpréteur* au nœud de jointure. Enfin, si l'outil Git est sélectionné (choix *Git* sélectionné), alors il faut remplacer l'outil SVN par l'outil Git.



(a) Première partie du VRM



(b) Deuxième partie du VRM

FIGURE 4.10 – Extrait du VRM de l'exemple illustratif

4.2.2 Dérivation d'un processus en fonction des exigences d'un projet

4.2.2.1 Résolution de la variabilité des exigences des projets (étape 4)

Approche Un utilisateur de la ligne de processus précédemment définie résout la variabilité des exigences des projets. Il sélectionne pour ce faire dans le RM de CVL les exigences pour un projet donné, parmi les exigences spécifiées dans le VAM.

Illustration La figure 4.11 présente un extrait du RM pour un processus de métamodélisation, avec uniquement *éditeur arborescent*, *interpréteur* et *SVN* comme choix sélectionnés. Les choix dérivés sont automatiquement résolus en fonction de la résolution des autres choix. Ici, seuls les choix dérivés *parallélisation*, *interpreteur parallèle* et *éditeur arborescent parallèle* sont sélectionnés.

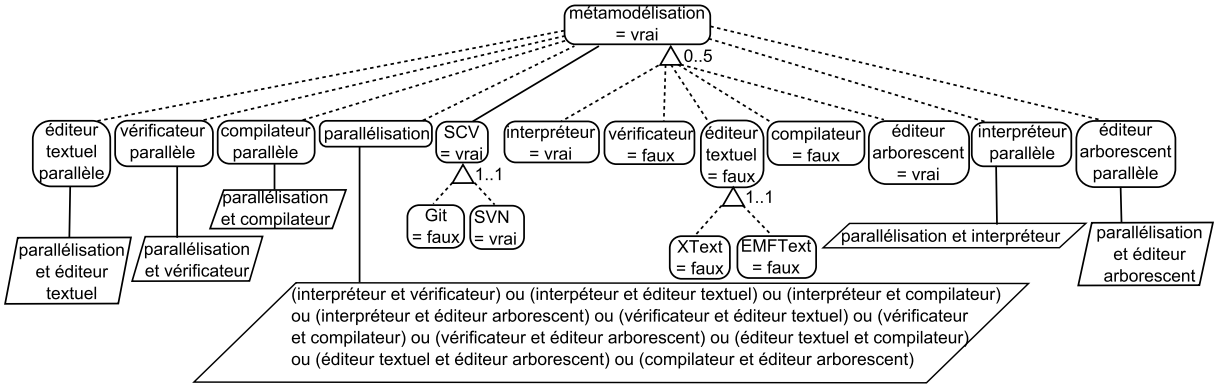


FIGURE 4.11 – Extrait du RM de l'exemple illustratif

4.2.2.2 Dérivation automatique de processus (étape 5)

Approche La dernière étape de notre approche consiste à dériver automatiquement le processus correspondant aux exigences d'un projet donné. Dans ce but, nous utilisons le moteur de dérivation de CVL afin de permettre à l'utilisateur de la ligne de processus de dériver un modèle de processus depuis le modèle de processus de base (étape 1), en fonction i) des exigences sélectionnées dans le RM (étape 4), et ii) de la réalisation de la variabilité capturée dans le VRM (étape 3). Cette étape produit un modèle de processus résolu. Si un projet a des besoins spécifiques qui ne se retrouvent pas dans d'autres projets, alors l'utilisateur de la ligne de processus peut adapter manuellement le modèle de processus résolu à ces besoins, à l'issue de la dérivation automatique de ce modèle de processus. En effet, il n'est pas utile de capitaliser des besoins spécifiques à un projet dans une ligne de processus si ces besoins ne se retrouvent pas dans d'autres projets.

Illustration Comme illustré par la figure 4.12, le processus résolu correspondant au RM de la figure 4.11 est un processus de métamodélisation avec les étapes de définition

d'un métamodèle, d'un éditeur arborescent, d'un interpréteur, et avec le SCV SVN.

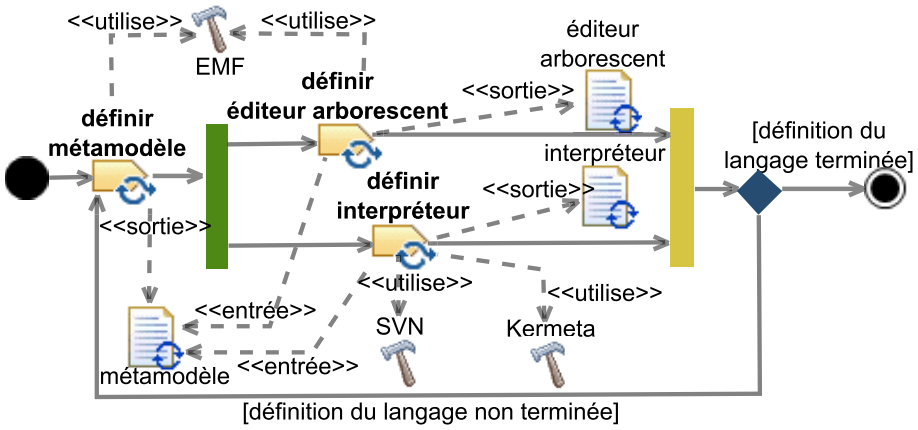


FIGURE 4.12 – Modèle de processus résolu

4.2.3 Exécution d'un processus résolu

La difficulté au moment de l'exécution d'un processus dérivé d'une ligne de processus réside dans le fait qu'en fonction des contraintes d'un projet, ce processus peut être exécuté alors que sa variabilité n'est que partiellement résolue. Par exemple, afin de gagner du temps, il est possible de démarrer l'exécution d'un processus de développement d'une application web en Java (par exemple en écrivant les spécifications fonctionnelles) alors que le framework pour l'IHM de l'application à développer (par exemple Struts [6], JSF [7] (Java Server Faces), Flex [8] ou GWT [9] (Google Web Toolkit)) n'a pas encore été choisi. Dans ce cas, l'automatisation de l'exécution du processus peut donner lieu à des erreurs à cause de la variabilité non résolue. En effet, si une partie d'un processus contient de la variabilité non résolue, alors :

- soit cette partie de processus est dans un état où elle n'est pas exécutable par un outil, car elle possède des éléments considérés comme invalides par cet outil (par exemple un flot de contrôle qui ne pointe vers rien) ;
- soit cette partie est dans un état exécutable, mais qui ne correspond pas à l'état dans lequel elle aurait dû être si la variabilité avait été résolue. Dans ce cas, le résultat de l'exécution ne sera pas celui attendu.

Afin de gérer les cas où la variabilité du processus en cours d'exécution n'est que partiellement résolue, notre approche consiste à demander à l'acteur courant du processus de résoudre la variabilité d'un fragment de processus dont la variabilité n'est que partiellement résolue au moment de l'exécution de ce fragment.

6. http://struts.apache.org/
7. http://www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html
8. http://www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html
9. http://www.gwtproject.org/

4.3 Discussion

Nous discutons dans cette section de la capacité de CVL à gérer la variabilité des processus en fonction de la variabilité des exigences des projets (section 4.3.1), de l'indépendance de CVL vis-à-vis des métamodèle de processus (section 4.3.2), de la validité du modèle de processus résolu (section 4.3.3), ainsi que des extensions possibles à la méthodologie proposée pour définir le modèle de processus de base (section 4.3.4).

4.3.1 Capacité de CVL à gérer la variabilité des processus en fonction de la variabilité des exigences des projets

Selon notre approche, le VAM capture la variabilité des exigences des projets, tandis que le VRM spécifie le lien entre cette variabilité des exigences et le modèle de processus de base. Le VRM permet de refléter directement la variabilité des exigences des projets sur les processus. Ces trois modèles (VAM, VRM et modèle de processus de base) constituent donc la définition d'une ligne de processus de développement logiciel. CVL permet ensuite à un utilisateur de cette ligne de processus i) de sélectionner les exigences pour un projet donné dans un RM et ii) de dériver automatiquement un modèle de processus depuis le modèle de processus de base en fonction de ce RM. CVL permet donc bien de gérer la variabilité des processus de développement logiciel en fonction de la variabilité des exigences des projets.

4.3.2 Indépendance de CVL vis-à-vis des métamodèles de processus

Actuellement, le seul lien entre CVL et les modèles métiers pour lesquels il permet de gérer la variabilité est fait par l'intermédiaire des points de variation, qui permettent de faire des références aux éléments du modèle de base. Ces références sont faiblement couplées aux éléments du modèle de base. En effet, CVL utilise une chaîne de caractères dans les éléments de modèle de type `ObjectHandle` et `LinkHandle` pour référencer les éléments du modèle de base. Dans son état actuel, CVL est donc bien indépendant du langage sur lequel il s'applique. Nous allons cependant voir dans la section suivante que le couplage faible entre CVL et les langages sur lesquels il s'applique peut être à l'origine d'erreurs dans le modèle de processus résolu.

4.3.3 Validité du modèle de processus résolu

Nous avons observé que l'opérateur de dérivation de CVL pouvait produire un modèle de processus résolu invalide (c'est-à-dire non conforme à son métamodèle), alors que les VAM, VRM et RM sont valides. Dans la suite de cette section, nous identifions les sources possibles de cette invalidité, et nous les classifions en fonction de la manière de les éviter.

Le premier type d'erreur que nous avons observé concerne les affectations (affectation d'attribut, paramétrable ou non) ou les substitutions (substitution de fin de lien,

substitution de fin de lien paramétrable, substitution d'objet, substitution d'objet paramétrable, substitution de fragment) qui ne respectent pas la compatibilité de type avec le modèle de base.

Ces erreurs constituent la première catégorie que nous avons identifiée. Elles ont lieu parce que CVL ne contraint pas la spécification du VRM en fonction du métamodèle auquel il s'applique. L'expression de contraintes génériques sur le métamodèle de CVL permettrait d'assurer la validité du VRM par rapport au métamodèle du modèle auquel il s'applique. Par exemple, le problème de compatibilité de type mentionné plus haut dans le contexte d'une substitution de fin de lien pourrait être évité en utilisant la contrainte suivante exprimée avec OCL (*Object Constraint Language*) [OMG10] sur le métamodèle de CVL :

```
1 context LinkEndSubstitution inv:
2   -- où 'find' résoud l'URI d'un élément de modèle, 'getRef' retourne la
       référence, dont le nom est passé en paramètre, de l'élément de modèle sur
       lequel est appelée la fonction, 'OclType' retourne le type de l'élément de
       modèle sur lequel il est appelé et 'isOfType' retourne vrai si le type sur
       lequel il s'applique est du même type que le type passé en paramètre.
3   find(self.replacementObject.MOFRef).OclType.isOfType(find(self.link.MOFRef).
       getRef(self.linkEndIdentifier).OclType)
```

Cette contrainte spécifie en effet que dans le cas d'une substitution de fin de lien, l'objet correspondant à la nouvelle valeur d'un lien doit être du même type que la référence que ce lien instancie.

Le second type d'erreur concerne le non-respect de la multiplicité d'une référence. Par exemple, lorsqu'une référence avec une borne inférieure n et une borne supérieure p est instanciée par m liens, avec $m < n$ ou $m > p$. Cela a lieu lorsqu'une substitution de fin de lien ou une substitution de fin de lien paramétrable (une existence de lien) crée (supprime) un lien, rendant ainsi le nombre de liens instanciant une référence supérieur (inférieur) à sa borne supérieure (inférieure). Cela a également lieu durant une substitution de fragment, car CVL permet de créer et de supprimer des liens entrants et sortants au fragment remplaçant et à ses copies, sans assurer le respect de la multiplicité des références dont ces liens sont instances. Finalement, ce type d'erreur peut aussi arriver lors de l'exécution d'un point de variation composite, quand le conteneur du conteneur configurable ne peut pas contenir une nouvelle instance du conteneur configurable. Afin de prévenir ce type d'erreur, certaines contraintes doivent être respectées au moment de la dérivation. Un nouveau lien ne doit pouvoir être instance d'une référence dont la borne supérieure est strictement supérieure à un que si le nombre de liens qui sont déjà instances de cette référence est strictement plus petit que la borne supérieure de cette référence. Un lien peut être supprimé seulement si la référence dont il est instance avant sa suppression a un nombre de liens strictement supérieur à sa borne inférieure. Un fragment de processus peut être dupliqué seulement si ses liens entrants peuvent également être dupliqués tout en assurant le respect de la multiplicité des références dont ils sont instances. Un point de variation composite peut être exécuté seulement si le conteneur du conteneur configurable peut contenir un conteneur configurable de plus.

Le troisième type d'erreur concerne le non-respect des propriétés d'un métamodèle

capturées à l'aide d'un langage de contraintes. Ce type d'erreur peut apparaître à la suite de l'exécution de n'importe quel type de point de variation.

Les second et troisième types d'erreur constituent la seconde catégorie d'erreurs que nous avons identifiée. Ces erreurs sont possibles parce que CVL ne contraint pas la spécification du VRM en fonction du modèle de base auquel il s'applique et en fonction du métamodèle de ce modèle de base. Une solution pour assurer les contraintes des second et troisième types d'erreur serait un outil générique d'analyse statique qui vérifierait avant la dérivation si ces contraintes sont satisfaites. Par générique, nous entendons indépendant du métamodèle du modèle de processus de base. Il est possible d'implémenter un tel outil en raisonnant sur le métamétamodèle MOF, commun à tous les modèles de processus de base, et non pas sur un métamodèle particulier. Si ces contraintes ne sont pas satisfaites, l'outil informerait alors l'utilisateur de la ligne de processus de la contrainte qui a été violée et du point de variation ayant introduit cette violation. L'expert processus et l'utilisateur de la ligne de processus devraient alors corriger l'erreur pour pouvoir commencer la dérivation.

La quatrième erreur apparaît quand le modèle de processus résolu est bien conforme à son métamodèle, mais est sémantiquement inconsistant. Un modèle de processus peut être sémantiquement inconsistant dans deux cas : soit lorsque ce modèle est trop permissif vis-à-vis d'un métier spécifique, soit lorsque ce modèle est incohérent. La figure 4.13 illustre le cas d'un modèle incohérent. En SPEM 2.0, une séquence de travail spécifie les dépendances au moment de l'exécution d'activités. Ici, la séquence de travail de type *finishToFinish* signifie que l'activité *B* peut finir quand l'activité *A* est terminée. La séquence de travail *finishToStart* signifie que l'activité *B* peut commencer quand l'activité *A* est terminée. Ici la séquence de travail de type *finishToFinish* est inutile car elle est automatiquement respectée si la séquence de travail de type *finishToStart* l'est. Afin d'éviter ce problème, on doit s'assurer qu'il n'y a pas d'inconsistances au niveau de la sémantique dans le modèle de processus résolu.

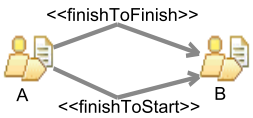


FIGURE 4.13 – Séquence de travail sur-spécifiée

Ce quatrième type d'erreur constitue la troisième catégorie que nous avons identifiée. L'origine de ces erreurs est l'utilisation d'un métamodèle de processus non adapté à un domaine métier ou permettant la création de modèles incohérents. Comme CVL ne contraint pas la spécification du VRM en fonction du modèle de base et de son métamodèle, les erreurs de la troisième catégories peuvent avoir lieu. Cependant, dans ce cas, les erreurs sont spécifiques à un métamodèle et ne peuvent pas être généralisées. Nous proposons deux solutions pour éviter ces erreurs. L'une préserve l'indépendance vis-à-vis du métamodèle de processus, l'autre non. La première solution serait de modifier le métamodèle du modèle de processus résolu afin qu'il ne permette plus de définir des instances incohérentes et qu'il satisfasse les contraintes d'un domaine métier spécifique. La deuxième solution serait d'implémenter un moteur de dérivation

spécifique au métamodèle de processus utilisé (ici SPEM). Il assurerait que durant la dérivation des contraintes spécifiques à ce métamodèle soient satisfaites, en plus d'effectuer les mêmes opérations que le moteur de dérivation générique de CVL. Dans le cas où une contrainte spécifique au métamodèle ne serait pas respectée, le moteur de dérivation informerait l'utilisateur de la ligne de processus de l'étape de dérivation qui a introduit cette erreur et de la contrainte qui a été violée. Le choix entre la première et la deuxième solution dépend de leur coût de mise en œuvre. En effet, dans le cas de la première solution, la modification du métamodèle de processus implique de modifier les outils le supportant (éditeurs, compilateurs, interpréteurs...). En contrepartie, on peut continuer à utiliser le moteur de dérivation générique de CVL. Cette solution préserve donc l'indépendance de CVL vis-à-vis du langage de modélisation de processus utilisé. La deuxième solution, étant donné qu'elle n'implique pas de modifier le métamodèle de processus, n'implique pas non plus de mettre à jour les outils supportant ce métamodèle. Elle impose en revanche de définir un moteur de dérivation spécifique à un métamodèle de processus, lorsque celui-ci n'est pas parfaitement adapté à un métier ou lorsqu'il permet la création d'instances incohérentes. Il incombe donc à chaque utilisateur de CVL de déterminer la solution qui lui convient le mieux, en fonction de ses ressources et des erreurs qu'il souhaite éviter.

En conclusion, même s'il est actuellement possible de dériver un modèle de processus résolu invalide, cela ne remet pas en cause l'indépendance actuelle de CVL vis-à-vis du métamodèle du modèle de base et du modèle résolu, et il existe des solutions à ce problème qui préservent cette indépendance.

4.3.4 Extension possible à la méthodologie pour modéliser le processus de base

La modélisation des éléments de processus externes peut conduire à des modèles qui ne sont pas conformes à leur métamodèle. En effet, comme les éléments de processus externes sont modélisés indépendamment d'un processus, certains ne respectent pas toutes les contraintes imposées par leur métamodèle. Par exemple, le modèle de la figure 4.8 n'est pas conforme au métamodèle des diagrammes d'activité UML. En effet, tous les flots de contrôle n'ont pas forcément à la fois une source et une cible, alors que le métamodèle des diagrammes d'activité UML l'impose. Puisque certains modeleurs de processus (par exemple, SPEM-Designer [10]) ne permettent pas de modéliser des éléments de processus invalides, l'application de la méthodologie que nous proposons pour définir le modèle de processus de base peut se révéler difficile. Nous proposons donc d'utiliser des approches (par exemple [RBJ07]) permettant de générer des éléments de modèles en fonction de contraintes prédéfinies afin de gérer ce cas. Il serait ainsi possible de générer les éléments de modèles que les modeleurs ne permettent pas de modéliser, et également de générer les éléments de modèles manquants pour les rendre valides. Ces éléments de modèle générés ne seraient bien entendu pas utilisés pour dériver des processus de la ligne de processus.

10. http://marketplace.eclipse.org/content/spem-designer-helios-version

4.4 Synthèse

Nous avons proposé dans ce chapitre une approche permettant de gérer la variabilité des processus de développement logiciel en fonction des exigences des projets, et ce quel que soit le langage de modélisation de processus utilisé pour peu qu'il se conforme au MOF. Notre approche s'appuie sur CVL afin de spécifier et de résoudre la variabilité dans les processus. Nous avons fait le choix de CVL car il préserve la séparation des préoccupations entre l'aspect gestion de la variabilité et l'aspect métier dont on souhaite gérer la variabilité. Il est donc possible de réutiliser les concepts et outils spécifiques à ces aspects, sans qu'ils soient impactés par des préoccupations externes. D'autre part, bien que CVL permette actuellement de dériver des modèles résolus invalides, il existe des solutions permettant de prévenir ce problème. Ces solutions ont également l'avantage de préserver l'indépendance vis-à-vis du domaine métier dont la variabilité est gérée. De plus, CVL permet de définir en intention les différents processus d'une famille. Il permet en effet, dans le modèle de processus de base, de ne définir qu'une seule fois les éléments de processus communs à plusieurs processus, et il fournit des mécanismes permettant d'intégrer ces éléments à un processus au moment de la dérivation. Cela facilite la maintenance d'une famille de processus puisque les redondances entre les différents processus sont évitées. Cela facilite également la réutilisation des fragments de processus communs à plusieurs processus d'une famille. D'autre part, notre approche préserve la séparation entre exigences et processus et elle permet de refléter directement la variabilité des exigences sur les processus, plutôt que de passer par une couche intermédiaire définissant la variabilité des processus. Enfin, la méthodologie que nous proposons pour définir le modèle de processus de base a l'avantage d'expliciter le processus d'une famille qui est le plus souvent utilisé.

Chapitre 5

Création de composants d'automatisation réutilisables

Nous avons vu au début de cette partie que la réutilisation des CA (Composants d'Automatisation, c'est-à-dire composants logiciels automatisant des TMR) était réalisée via la réutilisation des processus de développement logiciel auxquels ces CA sont liés. Chaque CA peut être lié à des processus différents ou à des unités de travail différentes d'un même processus, constituant ainsi des cas d'utilisation différents d'un CA. Nous présentons donc dans ce chapitre notre méthodologie fournissant un support à la création de CA réutilisables à travers tous leurs cas d'utilisation. Nous commençons par introduire, dans la section 5.1, l'exemple dont nous nous servirons pour illustrer cette méthodologie, qui est elle-même détaillée dans la section 5.2. Nous discutons cette méthodologie dans la section 5.3 et nous en faisons la synthèse dans la section 5.4.

Les travaux présentés dans ce chapitre ont fait l'objet d'une publication :

Emmanuelle Rouillé, Benoît Combemale, Olivier Barais, David Touzet and Jean-Marc Jézéquel. Improving Reusability in Software Process Lines. In Proceedings of the 39th Euromicro Conference on Software Engineering and Advanced Applications, SEAA '13, pages 90-94, 2013 [RCB+13a].

5.1 Exemple illustratif : automatisation de la configuration de l'espace de travail local d'un développeur

Nous présentons dans cette section l'exemple dont nous nous servirons pour illustrer la méthodologie fournissant un support à la création de CA réutilisables. Il s'agit d'un script PowerShell qui automatise la configuration de l'espace de travail local d'un

développeur. Ce script est utilisé pendant les projets de développement Java de l'entreprise Sodifrance. Il est exécuté au début de l'activité de développement par chaque développeur du projet. Il est également exécuté à chaque fois qu'un nouveau développeur intègre un projet de développement Java dont l'activité de développement est déjà commencée. La figure 5.1 montre un extrait de ce script PowerShell, sur lequel notre méthodologie n'a pas encore été appliquée (c'est-à-dire que le script est configuré pour être utilisé avec un projet particulier). Il prend en paramètre (ligne 1) le chemin de l'espace de travail local d'un développeur (*wsPath*), ainsi que les URL des dépôts contenant le code source de l'application en cours de développement (*sourceUrl*) et le code de *build* (*buildUrl*). Le code de *build* correspond à des ressources, autres que le code source, utiles à la configuration de l'espace de travail local d'un développeur. Le script automatise les étapes suivantes :

1. import, en utilisant SVN, du code source (ligne 10) et du code de *build* (ligne 13),
2. compilation Maven des projets Java et alimentation initiale du dépôt Maven local si nécessaire (ligne 18),
3. configuration Maven des projets Eclipse (ligne 19),
4. configuration Maven de l'espace de travail Eclipse (ligne 20),
5. import Buckminster des projets dans l'espace de travail Eclipse (à partir de la ligne 23).

5.2 La méthodologie

Nous constatons que les CA peuvent contenir de la variabilité. En effet, un même CA peut être lié à des unités de travail différentes appartenant à un même processus, représentant alors des *contextes d'utilisation* différents de ce CA. Par exemple, dans le cas d'un processus de métamodélisation comprenant des étapes de définition d'un métamodèle, d'un éditeur textuel et d'un éditeur arborescent, un CA pourrait automatiser la mise sous contrôle de version de code source lors de toutes ces étapes. De plus, une même unité de travail peut varier en fonction des exigences d'un projet (par exemple utilisation de Git au lieu de SVN), créant ainsi encore plus de contextes d'utilisation différents d'un CA. La variabilité au niveau des contextes d'utilisation d'un CA crée, quant à elle, de la variabilité au niveau des CA eux-mêmes. Par exemple, l'implémentation d'un CA qui fait de la mise sous contrôle de version sera différente selon le SCV utilisé, ou selon l'adresse du dépôt distant utilisé. Une difficulté supplémentaire concernant la réutilisation des CA est donc d'identifier les parties d'un CA qui varient et celles qui ne varient pas (c'est-à-dire identifier le *niveau de réutilisation* d'un CA), afin ensuite d'être capable d'implémenter des CA qui soient réutilisables à travers leurs différents contextes d'utilisation (en utilisant par exemple des mécanismes tels que le paramétrage, la modularisation...). Pour ce faire, nous présentons dans cette partie une méthodologie permettant d'améliorer l'identification du niveau de réutilisation de CA. Nous appelons cette méthodologie M4RAC (*Methodology for Reusable Automation Components*). Elle constitue la deuxième sous-contribution de cette thèse. La figure 5.2

```
1  Param([string] $wsPath, [string] $sourceUrl, [string] $buildUrl)
2
3  # Variable settings
4  $sourcePath = $wsPath + "/source"
5  $buildPath = $wsPath + "/build"
6
7  [...]
8
9  # Checkout of source code from URL  $sourceUrl to folder $sourcePath
10 svn checkout $sourceUrl $sourcePath
11
12 # Checkout of build code from URL $buildUrl to folder $buildPath
13 svn checkout $buildUrl $buildPath
14
15 # Run Maven commands to compile and configure Eclipse projects and to configure
       the workspace
16 $sourcePom = $sourcePath + "/pom.xml"
17 [...]
18 mvn compile -f $sourcePom
19 mvn eclipse:eclipse -f $sourcePom
20 Invoke-Expression ("mvn -D eclipse.workspace=" + $wsPath + " eclipse:configure-
       workspace")
21
22 # Run Buckminster commands to import projects into the workspace
23 $buckyBuild = $buildPath + "/buckybuild.properties"
24 [...]
25 $cQueryFile=$buildPath + "/org.eclipse.buckminster.myproject.build/buckminster/
       myproject-build.cquery"
26 buckminster import --properties $buckyBuild -data $wsPath $cQueryFile
27 [...]
```

FIGURE 5.1 – Extrait d'un script PowerShell qui automatise la configuration de l'espace de travail d'un développeur

illustre la partie de la contribution principale que nous détaillons ici, tandis que la figure 5.3 montre le processus de la méthodologie. Nous expliquons plus en détails ce processus dans la suite de cette section.

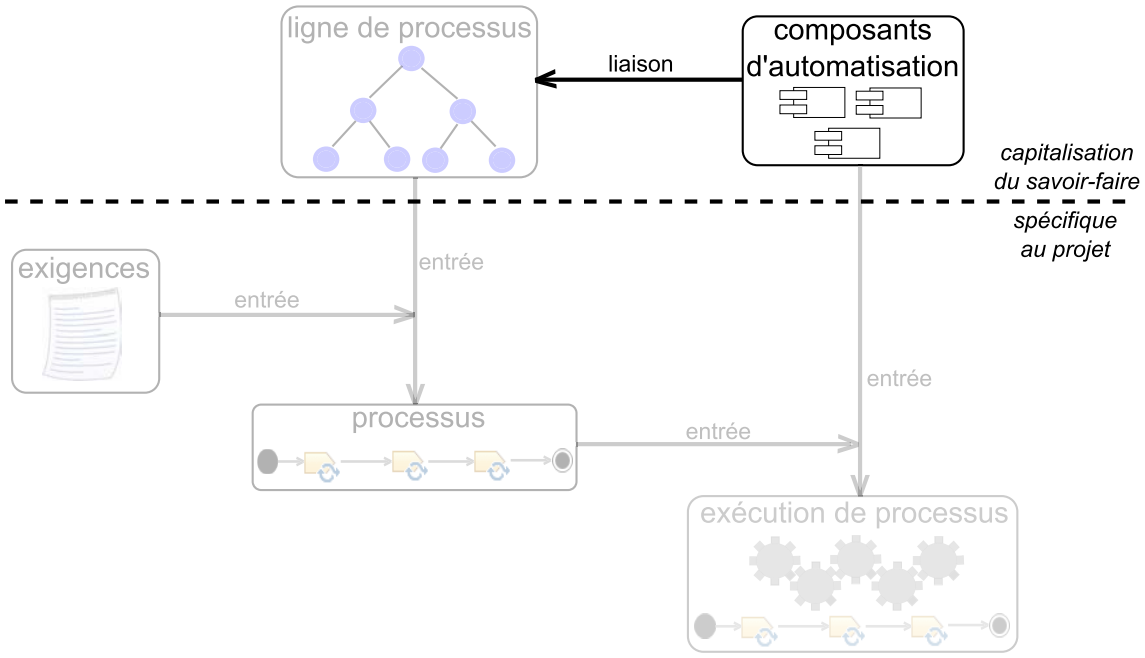


FIGURE 5.2 – Partie de la contribution principale réalisée par M4RAC

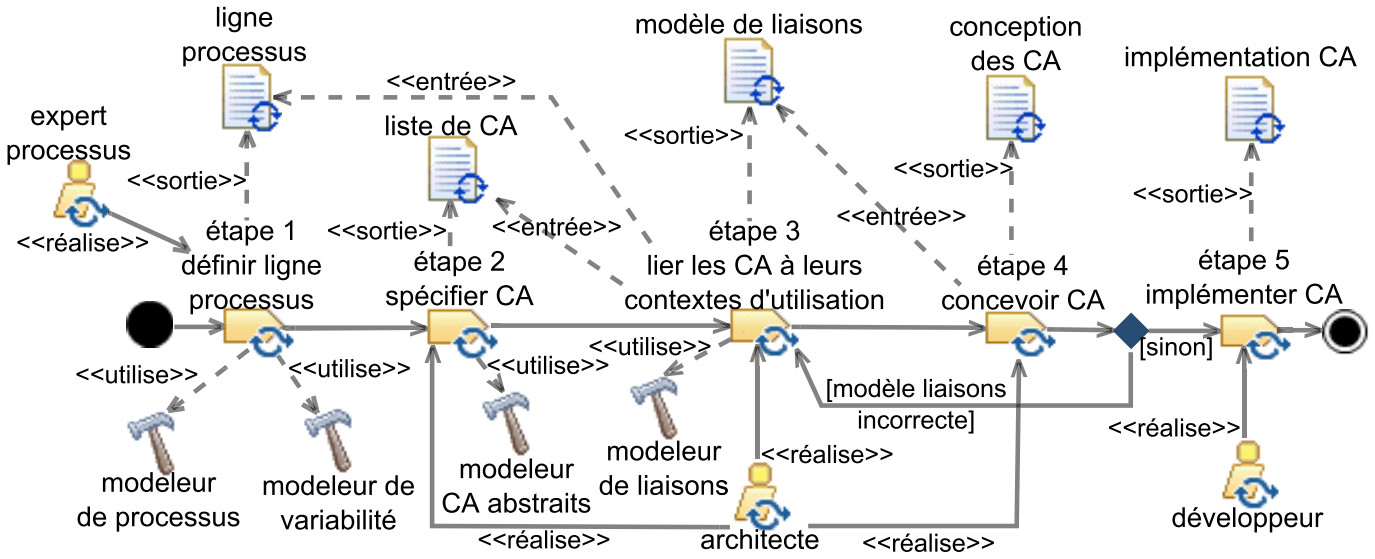


FIGURE 5.3 – Vue générale de M4RAC

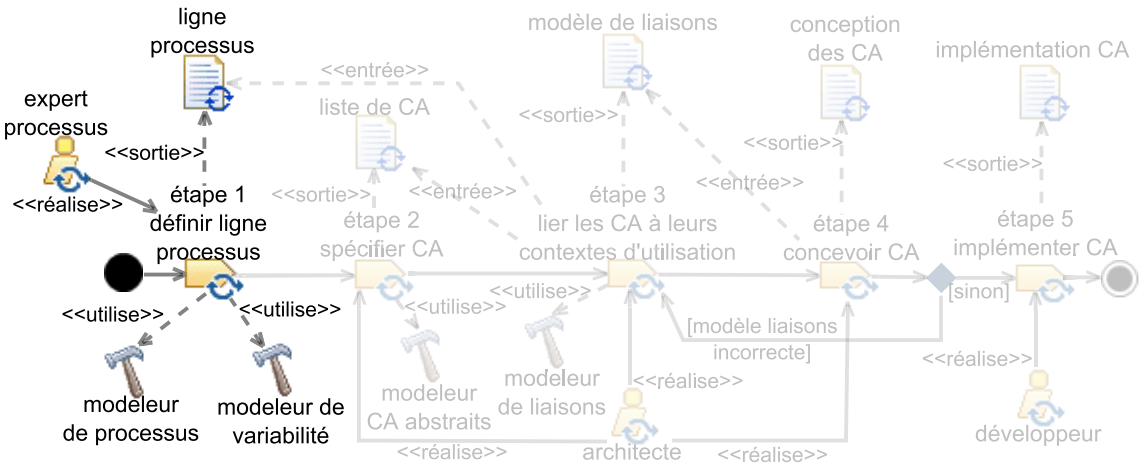


FIGURE 5.4 – Première étape de M4RAC

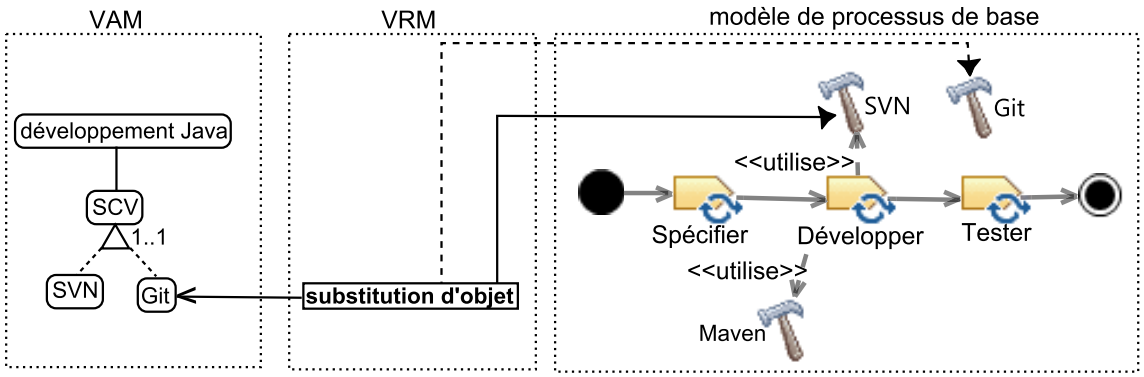


FIGURE 5.5 – Exemple simplifié de ligne de processus de développement Java

5.2.1 Définition d'une ligne de processus de développement logiciel (étape 1)

5.2.1.1 Méthodologie

La première étape de M4RAC, illustrée par la figure 5.4, implique comme rôle un expert processus, tout comme pour CVL4SP (chapitre 4). Cet expert processus définit une ligne de processus de développement logiciel en utilisant CVL4SP.

5.2.1.2 Illustration

La figure 5.5 montre un exemple simplifié de ligne de processus de développement Java. Dans le modèle de processus de base (à droite sur la figure 5.5), le processus le plus souvent utilisé est composé des tâches *spécifier*, *développer* et *tester*, et les outils *SVN* et *Maven* servent à réaliser la tâche *développer*. Le modèle de processus de base contient également un élément de processus externe, l'outil *Git*. Le VAM (à gauche

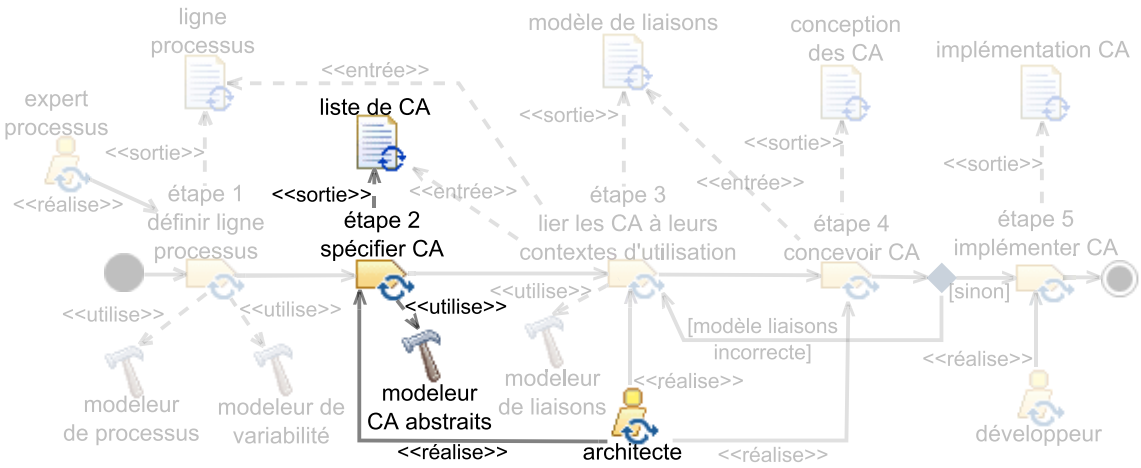


FIGURE 5.6 – Deuxième étape de M4RAC

sur la figure 5.5) spécifie qu'un SCV (correspondant sur la figure au choix *SCV*) est obligatoirement utilisé pendant un projet de développement Java (correspondant sur la figure au choix *développement Java*), et que ce SCV peut être soit Git (correspondant sur la figure au choix *Git*) soit SVN (correspondant sur la figure au choix *SVN*). Le VRM, au centre de la figure 5.5, spécifie quant à lui que si Git est choisi comme SCV, alors, dans le modèle de processus de base, l'outil *SVN* doit être remplacé par l'outil *Git*.

5.2.2 Spécification des composants d'automatisation (étape 2)

5.2.2.1 Méthodologie

La figure 5.6 met en évidence la deuxième étape de M4RAC. Cette étape implique comme rôle un architecte, c'est-à-dire une personne ayant l'expérience et le recul nécessaire pour identifier des TMR qui gagneraient à être automatisées, et capable de concevoir des CA automatisant ces TMR. Lors de la deuxième étape de M4RAC, l'architecte spécifie les CA abstraits qui sont utiles pour l'entreprise dans laquelle il travaille, à l'aide d'un modeleur de CA abstraits. Un CA abstrait est une définition conceptuelle d'un CA, où l'architecte spécifie uniquement qu'un CA existe et ce qu'il automatise, sans rentrer dans les détails de conception et d'implémentation. Durant cette étape, l'architecte ne se préoccupe pas de l'identification du niveau de réutilisation de ces CA, ni de leur conception et de leur implémentation. Il se préoccupe uniquement de la spécification, de manière abstraite, des CA existants ainsi que des CA futurs. Il détermine les CA futurs en identifiant les TMR qui gagneraient à être automatisées, et en supposant dans un premier temps que chacune de ces tâches sera automatisée par un CA.

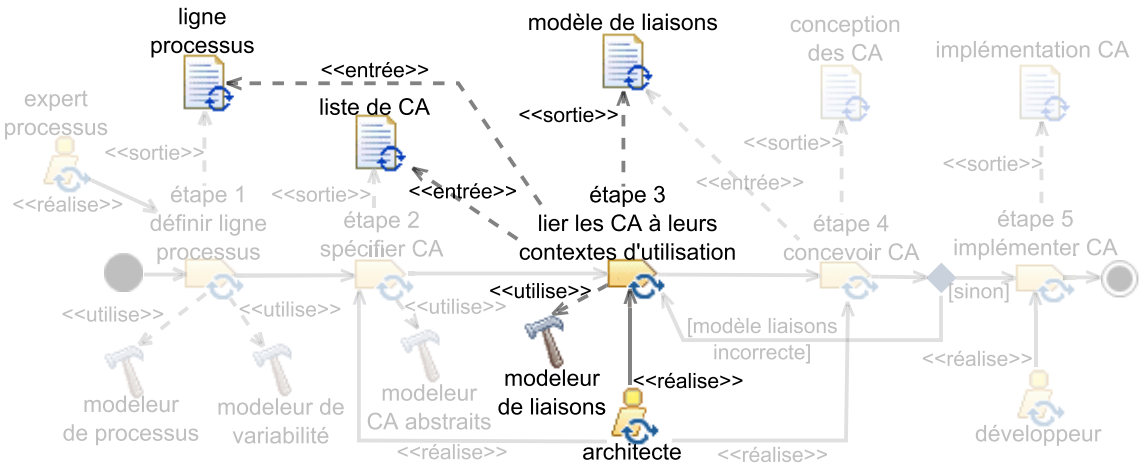


FIGURE 5.7 – Troisième étape de M4RAC

5.2.2.2 Illustration

Un exemple de CA spécifié à l'issue de la deuxième étape de M4RAC est le CA dont le but est de configurer l'espace de travail local d'un développeur.

5.2.3 Liaison entre les CA et leurs contextes d'utilisation (étape 3)

5.2.3.1 Méthodologie

Au moment de la troisième étape de M4RAC, mise en évidence par la figure 5.7, l'architecte modélise les liens entre les CA abstraits (spécifiés à l'étape 2) et les unités de travail (par exemple les tâches) de la ligne de processus qu'ils contribuent à automatiser. Il utilise pour ce faire un modeleur de liaisons, où ici le terme *liaisons* désigne les liens entre les CA et les unités de travail qu'ils automatisent. Si plusieurs CA contribuent à automatiser une même unité de travail, alors la liaison spécifie l'ordre dans lequel ces CA doivent s'exécuter. Si une unité de travail varie, alors la liaison spécifie quelle variante de cette unité de travail un CA contribue à automatiser. On considère qu'une unité de travail varie soit si l'unité de travail elle-même varie, soit si ce sont les éléments de modèle directement reliés à cette unité de travail (ex : outils, rôles, produits de travail, flots de contrôle) qui varient. Finalement, la liaison spécifie si un CA contribue à automatiser l'initialisation, l'exécution ou la finalisation d'une unité de travail. Définir une initialisation (une finalisation) pour une unité de travail est utile car cela permet d'empêcher l'oubli de cette initialisation (finalisation). En effet, l'initialisation (finalisation) est dans ce cas fortement couplée à l'unité de travail qu'elle initialise (finalise). Ce n'est pas le cas quand l'initialisation (la finalisation) est remplacée par une unité de travail qui a lieu avant (après) l'unité de travail initialisée (finalisée).

Cette étape permet donc d'identifier les contextes d'utilisation des CA. Plus précisément, ce sont les unités de travail liées à un CA, leurs ressources (c'est-à-dire les

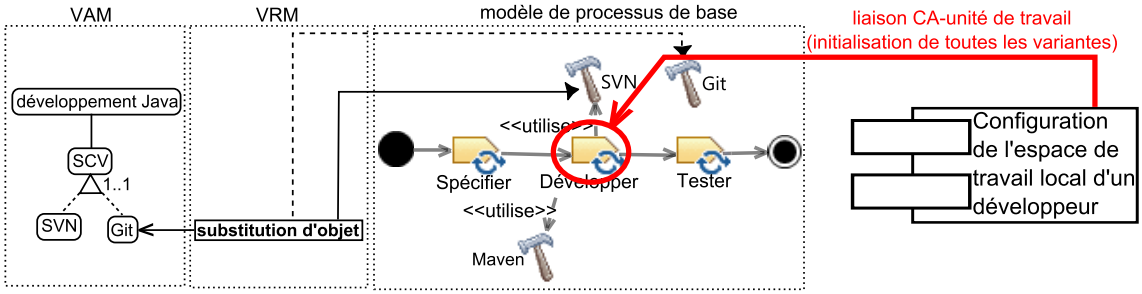


FIGURE 5.8 – Liaison entre le CA configurant l'espace de travail d'un développeur et la ligne de processus de la figure 5.5

outils, les rôles et les produits de travail), leurs itérations et leur séquencement qui définissent les contextes d'utilisation des CA. En effet, les variantes des unités de travail qu'un CA automatise sont ses contextes d'utilisation. Et c'est la spécification des variantes de ressources, itérations et séquencement associés à une unité de travail qui définit quelle variante d'unité de travail est considérée. De plus, nous verrons dans la prochaine partie de cette thèse que les unités de travail et leurs ressources fournissent les données (par exemple l'URL d'un dépôt) requises pour exécuter les CA.

5.2.3.2 Illustration

La figure 5.8 illustre la liaison entre le CA configurant l'espace de travail local d'un développeur et la ligne de processus de la figure 5.5. Le CA configurant l'espace de travail local d'un développeur contribue à l'automatisation de l'initialisation de toutes les variantes (c'est-à-dire en utilisant Git ou SVN) de la tâche *développer*. Cela permet donc d'identifier que ce CA a deux contextes d'utilisation : l'un lorsque SVN est utilisé comme SCV, et l'autre lorsque c'est Git qui est utilisé à la place de SVN.

5.2.4 Conception des composants d'automatisation (étape 4)

5.2.4.1 Méthodologie

Lors de la quatrième étape de M4RAC, mise en évidence par la figure 5.9, l'architecte conçoit les CA spécifiés de manière abstraite à l'étape 2. Pendant cette étape, le modèle de liaisons permet d'améliorer l'identification du niveau de réutilisation des CA. En effet, en réfléchissant à comment concevoir un CA de manière à ce qu'il soit réutilisable à travers ses différents contextes d'utilisation, l'architecte est capable de déterminer son niveau de réutilisation, c'est-à-dire les parties de ce CA qui varient et celles qui ne varient pas. Une fois cette information établie, l'architecte conçoit des CA réutilisables en s'appuyant sur des mécanismes tels que le paramétrage ou la modularisation.

En concevant les CA, l'architecte peut se rendre compte que le modèle de liaisons est incorrect. Par exemple, un CA peut contenir des parties inutiles pour certains de ses contextes d'utilisation. Il sera donc, dans ce cas, plus judicieux de diviser ce CA en plusieurs CA afin de pouvoir sélectionner uniquement les parties de ce CA utiles

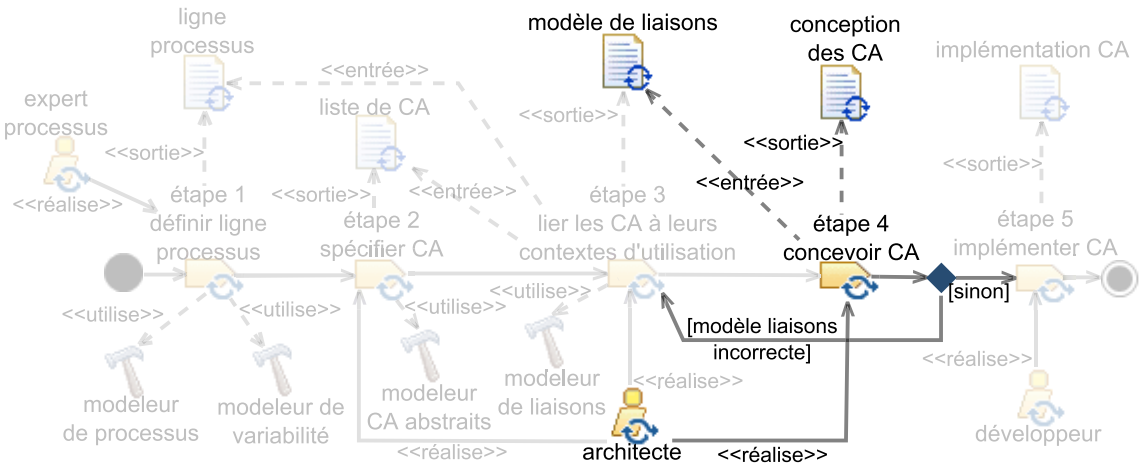


FIGURE 5.9 – Quatrième étape de M4RAC

pour chaque contexte d'utilisation. Dans ce cas, l'architecte revient à l'étape 3 afin de corriger le modèle de liaisons.

5.2.4.2 Illustration

Comme abordé dans la section 5.1, la configuration de l'espace de travail local d'un développeur se compose de 5 étapes :

1. import du code source et du code de *build* depuis un dépôt distant,
2. compilation Maven des projets Java et alimentation initiale du dépôt Maven local si nécessaire,
3. configuration Maven des projets Eclipse,
4. configuration Maven de l'espace de travail Eclipse,
5. import Buckminster des projets dans l'espace de travail Eclipse.

Le CA configurant l'espace de travail local d'un développeur doit donc réaliser ces 5 étapes. Or, l'étape 3 de M4RAC a permis de déterminer que ce CA pouvait être utilisé dans deux contextes différents : avec Git ou avec SVN. Cela permet donc de déterminer le niveau de réutilisation de ce CA, à savoir que ses parties dépendantes d'un SCV (étape 1 de la configuration de l'espace de travail local d'un développeur) peuvent varier tandis que les autres non (étapes 2 à 5 de la configuration de l'espace de travail local d'un développeur). L'architecte doit donc assurer que l'étape 1 soit découplée des étapes 2 à 5 de la configuration de l'espace de travail local d'un développeur, de manière à pouvoir réutiliser les étapes 2 à 5 indépendamment du SCV. Une manière de réaliser ce découplage est d'avoir trois CA : deux CA réalisant l'étape 1, l'un avec SVN (CA 1) et l'autre avec Git (CA 2), et un CA réalisant les étapes 2 à 5 (CA 3). L'initialisation de la tâche de développement serait alors effectuée par l'exécution des CA 1 puis 3 lorsque le SCV est SVN, et par l'exécution des CA 2 puis 3 lorsque le

SCV est Git. Le modèle de liaisons réalisé lors de l'étape précédente de M4RAC est également mis à jour, afin de refléter cette conception, comme illustré par la figure 5.10.

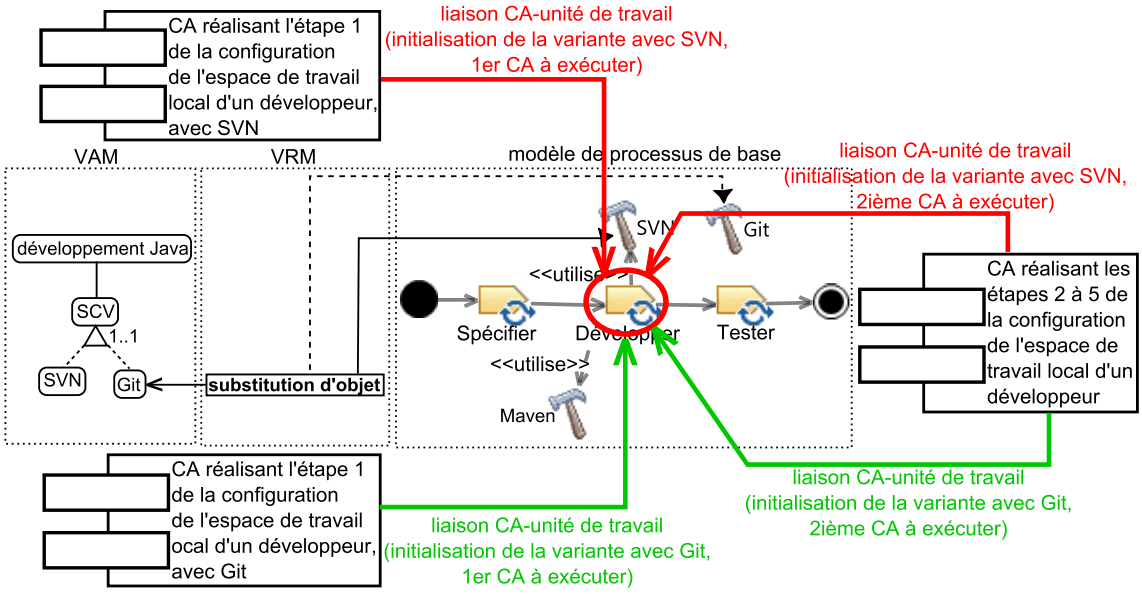


FIGURE 5.10 – Liaisons entre les CA permettant de configurer l'espace de travail d'un développeur et la ligne de processus de la figure 5.5

5.2.5 Implémentation des composants d'automatisation (étape 5)

5.2.5.1 Méthodologie

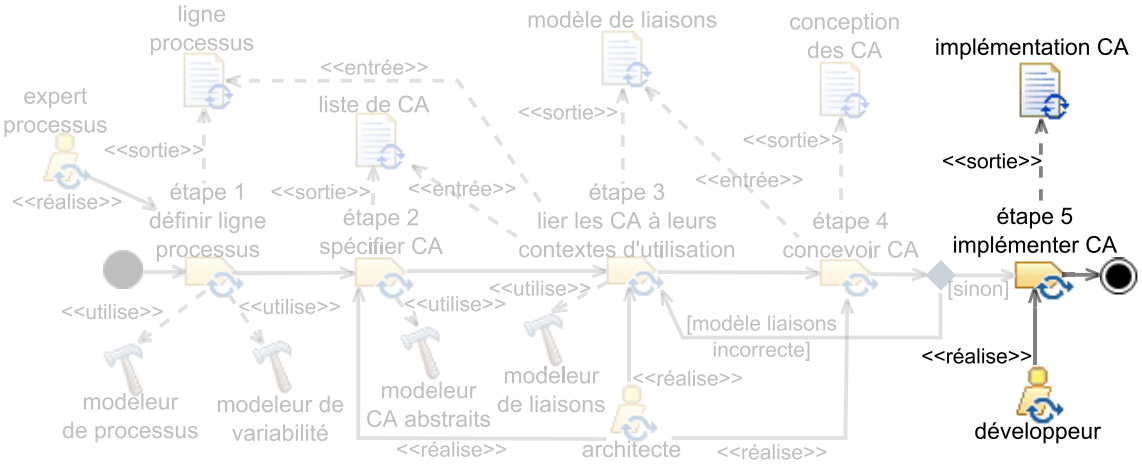


FIGURE 5.11 – Cinquième étape de M4RAC

Finalement, lors de la dernière étape de M4RAC, mise en évidence par la figure 5.11,

un développeur implémente les CA en fonction de leur conception (étape 4 de M4RAC).

5.2.5.2 Illustration

Le script PowerShell de la figure 5.1 correspond à une implémentation de la configuration de l'espace de travail local d'un développeur, dans le cas où SVN est utilisé comme SCV. Il sert de base à la création des trois CA conçus lors de l'étape 4 de M4RAC. Les scripts des figures 5.12, 5.13 et 5.14 correspondent respectivement à l'implémentation des CA 1, 2 et 3 identifiés précédemment.

```
1  Param([string] $wsPath, [string] $sourceUrl, [string] $buildUrl)
2
3  # Variable settings
4  $sourcePath = $wsPath + "/source"
5  $buildPath = $wsPath + "/build"
6
7  [...]
8
9  # Checkout of source code from URL  $sourceUrl to folder $sourcePath
10 svn checkout $sourceUrl $sourcePath
11
12 # Checkout of build code from URL $buildUrl to folder $buildPath
13 svn checkout $buildUrl $buildPath
```

FIGURE 5.12 – Extrait du CA automatisant l'étape 1 de la configuration de l'espace de travail d'un développeur, avec SVN

```
1  Param([string] $wsPath, [string] $sourceUrl, [string] $buildUrl)
2
3  # Variable settings
4  $sourcePath = $wsPath + "/source"
5  $buildPath = $wsPath + "/build"
6
7  [...]
8
9  # Checkout of source code from URL  $sourceUrl to folder $sourcePath
10 git clone $sourceUrl $sourcePath
11
12 # Checkout of build code from URL $buildUrl to folder $buildPath
13 git clone $buildUrl $buildPath
```

FIGURE 5.13 – Extrait du CA automatisant l'étape 1 de la configuration de l'espace de travail d'un développeur, avec Git

5.3 Discussion

Nous discutons dans cette partie M4RAC. Cette discussion porte sur ses bénéfices, ses limitations, ainsi que ses inconvénients.

```
1  Param([string] $wsPath)
2
3  # Variable settings
4  $sourcePath = $wsPath + "/source"
5  $buildPath = $wsPath + "/build"
6
7  # Run Maven commands to compile and configure Eclipse projects and to configure
       the workspace
8  $sourcePom = $sourcePath + "/pom.xml"
9  [...]
10 mvn compile -f $sourcePom
11 mvn eclipse:eclipse -f $sourcePom
12 Invoke-Expression ("mvn -D eclipse.workspace=" + $wsPath + " eclipse:configure-
       workspace")
13
14 # Run Buckminster commands to import projects into the workspace
15 $buckyBuild = $buildPath + "/buckybuild.properties"
16 [...]
17 $cQueryFile=$buildPath + "/org.eclipse.buckminster.myproject.build/buckminster/
       myproject-build.cquery"
18 buckminster import --properties $buckyBuild -data $wsPath $cQueryFile
19 [...]
```

FIGURE 5.14 – Extrait du CA automatisant les étapes 2 à 5 de la configuration de l'espace de travail d'un développeur

5.3.1 Bénéfices

Notre méthodologie fournit deux bénéfices principaux pour identifier le niveau de réutilisation de CA.

Le premier bénéfice de notre méthodologie est qu'elle aide l'architecte à identifier les différents contextes d'utilisation des CA. C'est parce qu'elle permet de spécifier les différents contextes d'utilisation des CA. En effet, chez Sodifrance, nous avons répertorié au moins 512 processus de développement Java possibles en pratique, identifiés suite à l'interview d'un chef de projet. Il est bien sûr impossible pour un humain d'avoir en tête ces différents processus et les différents contextes d'utilisation de tous les CA.

Le second bénéfice de notre méthodologie est qu'elle empêche l'architecte de considérer de la variabilité inutile. Cela prévient donc la conception et l'implémentation de CA inutiles. Le fait que M4RAC permette d'expliciter la variabilité des processus ainsi que les contextes d'utilisation des CA est la raison de ce bénéfice. Par exemple, dans le cas d'un CA en charge d'automatiser la mise sous contrôle de version du code source d'un projet, l'architecte pourrait concevoir trois variantes de ce CA : une utilisant CVS, une autre utilisant SVN et encore une autre utilisant Git. En effet, l'architecte pourrait déjà avoir utilisé ces trois SCV sur des projets différents. Cependant, l'architecte pourrait ne pas être au courant que CVS n'est plus utilisé (dans l'hypothèse où il est possible de déterminer que c'est le cas). Dans ce cas, implémenter un CA automatisant la mise sous contrôle de version du code source d'un projet en utilisant CVS serait une perte de temps. En revanche, si CVS n'est plus utilisé, alors il n'apparaîtra pas dans la ligne de processus comme un SCV possible. De ce fait, en appliquant notre méthodologie,

l'architecte aurait réalisé que CVS n'était plus utilisé. Il n'aurait alors pas conçu un CA inutile.

5.3.2 Une limitation

Une limitation de notre méthodologie est que son efficacité dépend des langages utilisés pour définir la ligne de processus. En effet, si ces langages ne permettent pas de capturer certaines informations concernant les contextes d'utilisation des CA (par exemple certains métamodèles de processus ne permettent pas de capturer une définition d'outil), alors l'architecte ne peut dans ce cas pas s'appuyer sur la ligne de processus pour identifier tous les contextes d'utilisation des CA. L'architecte ne peut alors s'appuyer que sur sa propre connaissance pour identifier les contextes d'utilisation qui ne sont pas capturés par la ligne de processus. Ceci est moins efficace que de s'appuyer sur la ligne de processus. En effet, l'architecte peut oublier des contextes d'utilisation ou peut considérer des contextes d'utilisation inutiles.

5.3.3 Un inconvénient

Un inconvénient de M4RAC est qu'il est difficile d'identifier les contextes d'utilisation des CA. En effet, avec la définition en intention des processus, il est difficile pour un humain de visualiser explicitement chacun des processus de la ligne. Par exemple, l'architecte peut avoir des difficultés à visualiser ce qui se passe avant et après une variante d'unité de travail, et donc il peut avoir des difficultés à déterminer à quelles unités de travail il est pertinent d'associer un CA. La définition de processus en extension n'est pas une solution satisfaisante à ce problème. En effet, plus le nombre de processus est élevé, plus il est tout aussi difficile pour un humain d'identifier les différents contextes d'utilisation des CA. Une solution serait d'avoir un outil support à l'identification des contextes d'utilisation des CA (étape 3 de M4RAC). Celui-ci pourrait par exemple calculer et afficher les différents processus d'une famille en fonction de leur spécification en intention, et afficher également les CA associés aux unités de travail de ces différents processus, au fur et à mesure de leur association. Afin de limiter le nombre de processus affichés, une idée serait de définir des contextes d'utilisation de CA similaires, et donc de n'afficher qu'un seul de ces contextes. De plus, cet outil pourrait vérifier la consistance entre des pré et post-conditions associées aux CA, afin de détecter d'éventuelles erreurs au moment de l'association d'un CA à une unité de travail.

5.4 Synthèse

Nous avons proposé, dans ce chapitre, une méthodologie qui permet d'améliorer l'identification du niveau de réutilisation d'un CA. Le principe de cette méthodologie consiste à lier chaque CA aux unités de travail qu'il contribue à automatiser dans une ligne de processus. Cela permet d'identifier les différents contextes d'utilisation de chacun de ces CA. Ensuite, en réfléchissant à la conception d'un CA afin qu'il soit

réutilisable à travers ses différents contextes d'utilisation, il est possible d'identifier le niveau de réutilisation de ce CA. Cette méthodologie permet d'expliciter les différents contextes d'utilisation d'un CA. De ce fait, cela permet à l'architecte en charge de la conception des CA de ne pas oublier certains contextes d'utilisation et de ne pas prendre en compte des contextes d'utilisation inutiles. Une limitation de cette méthodologie en revanche est que son efficacité dépend des langages utilisés pour définir la ligne de processus. En effet, les contextes d'utilisation que la ligne de processus peut capturer dépendent des informations que les langages utilisés pour définir la ligne de processus permettent de capturer. D'autre part, avoir un outil permettant d'expliciter les différents processus d'une ligne ainsi que les CA déjà associés à ces processus aiderait l'architecte à identifier les contextes d'utilisation des CA.

Troisième partie

Outillage et application

Dans cette partie, nous commençons par présenter dans le chapitre 6 l'outil que nous avons développé afin de démontrer la faisabilité de l'approche proposée dans cette thèse. Ensuite, dans le chapitre 7, nous appliquons cet outil à une famille de processus de métamodélisation ainsi qu'à une famille de processus de développement web Java issue de Sodifrance.

Chapitre 6

Outil pour la gestion de la variabilité et l'automatisation des processus

Ce chapitre traite de l'outil développé comme support à l'approche proposée dans cette thèse. Nous commençons par introduire, en section 6.1, l'exemple dont nous nous servons pour présenter cet outil. La vue générale de l'outil est présentée en section 6.2 et son implémentation est décrite en section 6.3. Nous présentons, dans la section 6.4, des éléments de méthodologie relatifs à l'utilisation de l'outil, ainsi que des extensions possibles. Enfin, nous faisons la synthèse de ce chapitre dans la section 6.5.

6.1 Exemples de tâches manuelles récurrentes

Cette section introduit les exemples de TMR (Tâches Manuelles Récurrentes) utilisés pour illustrer l'utilisation de l'outil développé dans cette thèse. Ces TMR, issues de l'exécution de processus de la famille des processus de métamodélisation (présentée dans la section 4.1), gagneraient à être automatisées. Elles correspondent à la manière de réaliser des processus de métamodélisation dans l'équipe de recherche Triskell. Il existe d'autres manières de réaliser des processus de métamodélisation qui ne sont pas traitées ici (utilisation d'un outil autre que Kermeta, utilisation différente d'EMFText, etc.).

TMR	Unité de travail pendant laquelle la TMR a lieu	Étape de l'unité de travail	Itération
Création projet EMF et fichier Ecore vide	déf. métamodèle	initialisation	première
Validation métamodèle	déf. métamodèle	finalisation	toutes

TMR	Unité de travail pendant laquelle la TMR a lieu	Étape de l'unité de travail	Itération
Création fichier contenant la syntaxe concrète du métamodèle	déf. éditeur arborescent	initialisation	première
Mise à jour fichier contenant la syntaxe concrète du métamodèle	déf. éditeur arborescent	initialisation	à partir de la seconde
Génération éditeur arborescent	déf. éditeur arborescent	finalisation	toutes
Génération syntaxe textuelle	déf. éditeur textuel avec EMFText	initialisation	première
Génération éditeur textuel	déf. éditeur textuel avec EMFText	finalisation	toutes
Création projet XText	déf. éditeur textuel avec XText	initialisation	première
Génération éditeur textuel	déf. éditeur textuel avec XText	finalisation	toutes
Création projet Kermeta initilisé avec un appel à une fonction Kermeta qui vérifie qu'un modèle respecte bien les contraintes définies sur son métamodèle	déf. vérificateur	initialisation	première
Création projet Kermeta et application patron de conception interpréteur au métamodèle	déf. interpréteur	initialisation	première
Application patron de conception interpréteur aux nouvelles métaclasses du métamodèle	déf. interpréteur	initialisation	à partir de la seconde
Mise sous contrôle de version interpréteur	déf. interpréteur	finalisation	première
Propagation, sur un dépôt partagé, du code source de l'interpréteur	déf. interpréteur	finalisation	toutes
Création projet Kermeta et application patron de conception visiteur au métamodèle	déf. compilateur	initialisation	première
Application patron de conception visiteur aux nouvelles métaclasses du métamodèle	déf. compilateur	initialisation	à partir de la seconde

TABLE 6.1 – Exemples de TMR réalisées durant un processus de métamodélisation

Nous avons identifié 16 exemples de TMR. Le tableau 6.1 les répertorie et précise pour chacune d'elle l'unité de travail, son étape (initialisation, exécution ou finalisation), ainsi que l'itération de processus concernées.

Deux TMR sont liées à la définition d'un métamodèle. La première consiste à créer un projet EMF ainsi qu'un fichier Ecore vide, qui contiendra plus tard la définition d'un métamodèle. Cette TMR a lieu au moment de l'initialisation de la définition d'un métamodèle, lors de la première itération. La seconde TMR consiste à valider un métamodèle et a lieu au moment de la finalisation de la définition d'un métamodèle, à chaque itération.

Trois autres TMR sont liées à la définition d'un éditeur arborescent. Celle-ci est initialisée par la création d'un fichier destiné à contenir la syntaxe concrète du métamodèle, lors de la première itération. La mise à jour de ce fichier a lieu lors des itérations suivantes, toujours à l'initialisation. À chaque itération, la définition d'un éditeur arborescent est finalisée par la génération de l'éditeur en lui-même.

Quatre TMR sont liées à la définition d'un éditeur textuel. Si celui-ci est défini avec EMFText, alors, dans le cadre de cet exemple, la TMR consistant à générer la syntaxe textuelle est réalisée à l'initialisation, lors de la première itération. Si l'éditeur textuel est défini avec XText, alors la TMR consistant à créer un projet XText est réalisée, à l'initialisation et lors de la première itération également. Dans les deux cas, la définition d'un éditeur textuel se termine par la TMR consistant à générer l'éditeur en lui-même.

Une TMR concerne la définition d'un vérificateur. Elle consiste à créer un projet Kermeta et à l'initialiser avec un appel à une fonction Kermeta vérifiant qu'un modèle respecte bien les contraintes définies sur son métamodèle. Elle a lieu au moment de l'initialisation, lors de la première itération.

Quatre TMR concernent la définition d'un interpréteur. Au moment de l'initialisation, lors de la première itération, une TMR crée un projet Kermeta et applique le patron de conception interpréteur à chaque méta-classe du métamodèle. Lors des itérations suivantes, la définition d'un interpréteur est initialisée par une TMR appliquant le patron de conception interpréteur seulement aux nouvelles méta-classes du métamodèle. Le cas des méta-classes du métamodèle supprimées ou modifiées n'est pas traité dans cet exemple. Lors de la première itération, la définition d'un interpréteur est finalisée par la TMR consistant à mettre l'interpréteur sous contrôle de version. De plus, à chaque itération, une TMR de propagation du code de l'interpréteur sur un dépôt distant a également lieu au moment de la finalisation.

Enfin, deux TMR sont liées à la définition d'un compilateur. La première consiste à créer un projet Kermeta et à appliquer le patron de conception visiteur à chaque méta-classe du métamodèle. Elle initialise la définition d'un compilateur et n'a lieu que lors de la première itération. La deuxième TMR applique le patron de conception visiteur seulement aux nouvelles méta-classes du métamodèle. Elle initialise la définition d'un compilateur à partir de la deuxième itération. Tout comme pour la définition d'un interpréteur, le cas des méta-classes du métamodèle supprimées ou modifiées n'est pas non plus traité dans cet exemple.

D'autres TMR non spécifiques au processus de métamodélisation, comme la configuration de l'environnement de développement, la configuration de l'environnement

d'intégration continue ou la configuration du schéma de compilation, peuvent également être réalisées dans ce contexte.

6.2 Vue générale de l'outil

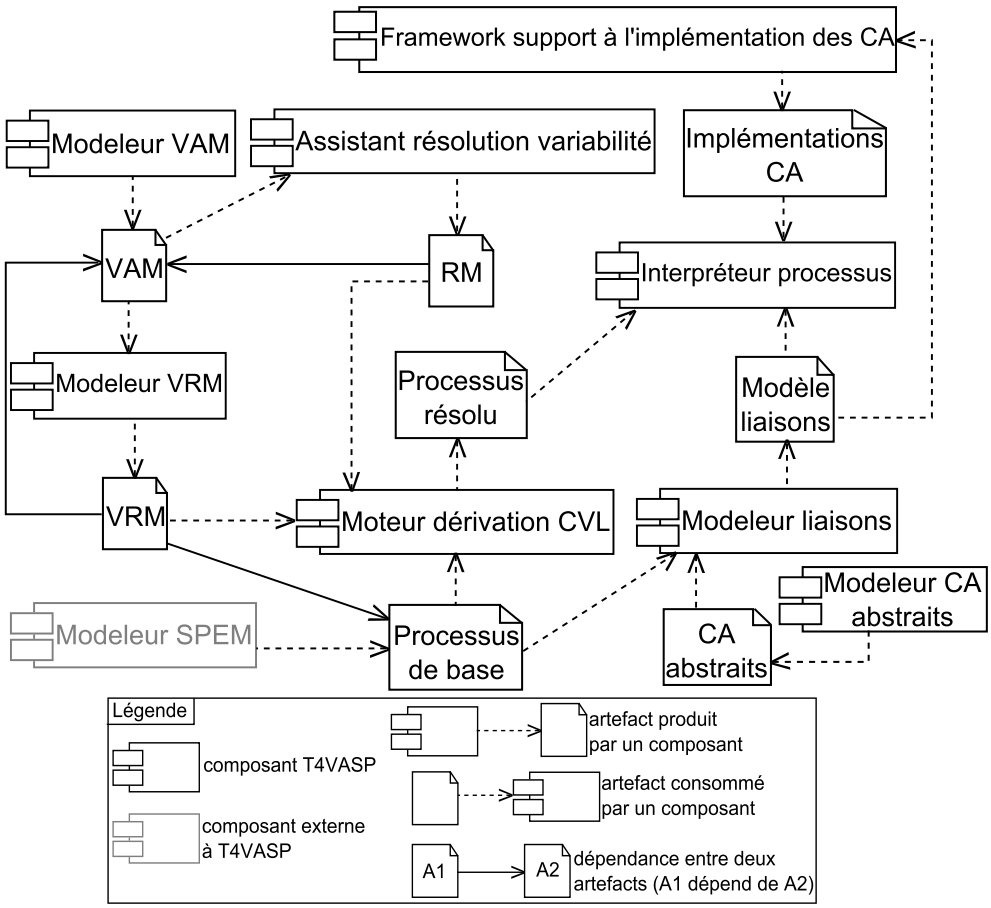


FIGURE 6.1 – Vue générale de l'architecture de T4VASP

T4VASP (*Tool for Variability management and Automation of Software Processes*) est l'outil que nous avons développé pour valider l'approche proposée dans cette thèse . La figure 6.1 illustre ses composants :

- un modeleur de VAM (*Variability Abstraction Model*, modèle spécifiant la variabilité (section 2.3.2 et chapitre 4)),
- un modeleur de VRM (*Variability Realization Model*, modèle définissant comment dériver un modèle résolu du modèle de base, en fonction de la résolution de la variabilité (section 2.3.2 et chapitre 4)),
- un assistant à la résolution de la variabilité,
- un moteur de dérivation CVL,

- un modeleur de CA (Composants d'Automatisation, composants logiciels automatisant des TMR (partie II)) abstraits,
- un modeleur de liaisons,
- un *framework* supportant l'implémentation des CA
- et un interpréteur de processus.

Les modeleurs de VAM et de VRM permettent respectivement de produire des VAM et des VRM. L'assistant à la résolution de la variabilité permet de résoudre la variabilité d'un VAM (donc permet de sélectionner les exigences pour un projet particulier) et produit un RM (*Resolution Model*, modèle permettant de résoudre la variabilité définie dans un VAM (section 2.3.2 et chapitre 4)). Le moteur de dérivation de CVL permet de dériver un modèle de processus résolu, en fonction d'un modèle de processus de base, d'un VRM et d'un RM. Le modèle de processus de base est produit à l'aide d'un modeleur de processus SPEM que nous réutilisons. Le modeleur de CA abstraits permet, comme son nom l'indique, de spécifier des CA abstraits, qui sont des définitions conceptuelles de CA (cf. section 5.2.2). Le modeleur de liaisons permet quant à lui de lier les CA abstraits i) aux unités de travail qu'ils contribuent à automatiser dans une ligne de processus et ii) à leur implémentation. Le *framework* support à l'implémentation des CA offre un cadre technique pour l'implémentation des CA. Enfin, l'interpréteur de processus permet d'exécuter un processus (obtenu par dérivation d'une ligne de processus), et de lancer au fur et à mesure de son exécution les CA qui l'automatisent. Nous détaillons ces composants dans la suite de cette section, en les présentant en fonction des parties de l'approche de cette thèse qu'ils supportent.

6.2.1 Définition d'une ligne de processus

Les modeleurs de VAM et de VRM supportent la définition d'une ligne de processus de développement logiciel, comme illustré par la figure 6.2. L'implémentation de ces modeleurs consiste à développer des éditeurs permettant de créer des VAM et des VRM conformes au métamodèle de la spécification de CVL. Des modeleurs autorisant la définition de VAM et de VRM dans des modèles distincts ont de plus l'avantage de permettre la réutilisation de ces modèles indépendamment les uns des autres. Ce besoin apparaît par exemple lorsqu'une même ligne de processus est définie avec deux langages de modélisation de processus différents, suite par exemple à un changement de modeleur dans l'entreprise. Dans ce cas, le même VAM est réutilisé pour spécifier la variabilité de la ligne de processus, quel que soit le langage de modélisation de processus utilisé, alors que des VRM différents sont utilisés pour chaque langage de modélisation de processus.

6.2.2 Définition des CA

Le modeleur de CA abstraits, le modeleur de liaisons ainsi que le *framework* support à l'implémentation des CA contribuent à la définition des CA. Plus précisément, ils contribuent à la spécification de CA abstraits, à leur liaison à la ligne de processus et à leur implémentation. La figure 6.3 illustre la partie de l'approche proposée dans cette

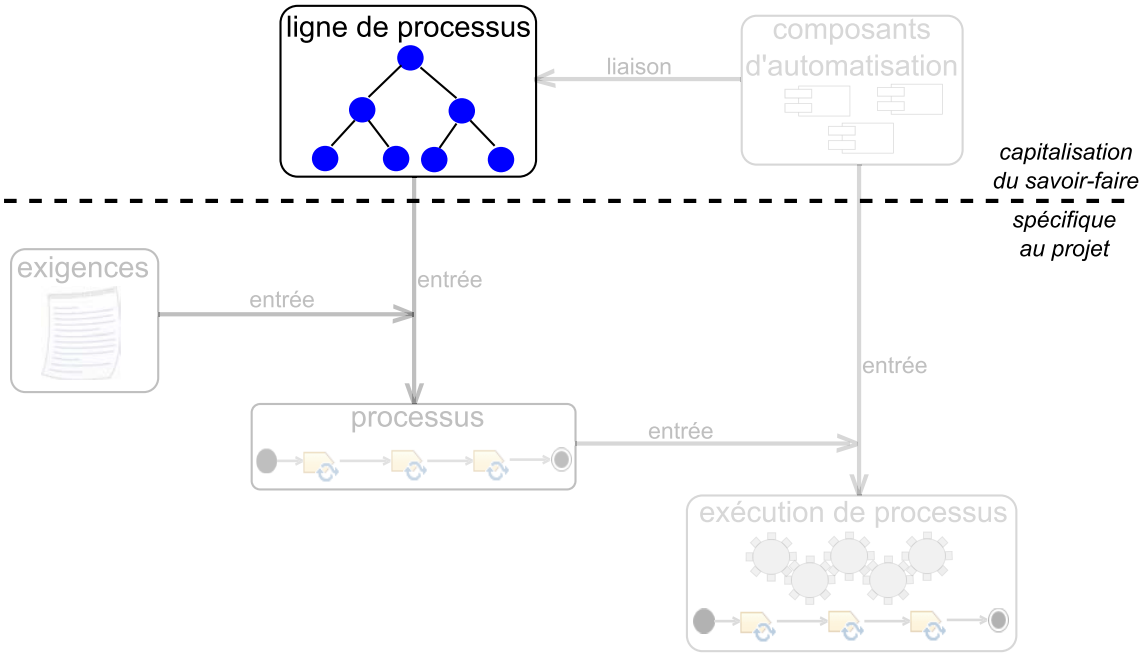


FIGURE 6.2 – Partie de l'approche supportée par les modeleurs de VAM et de VRM

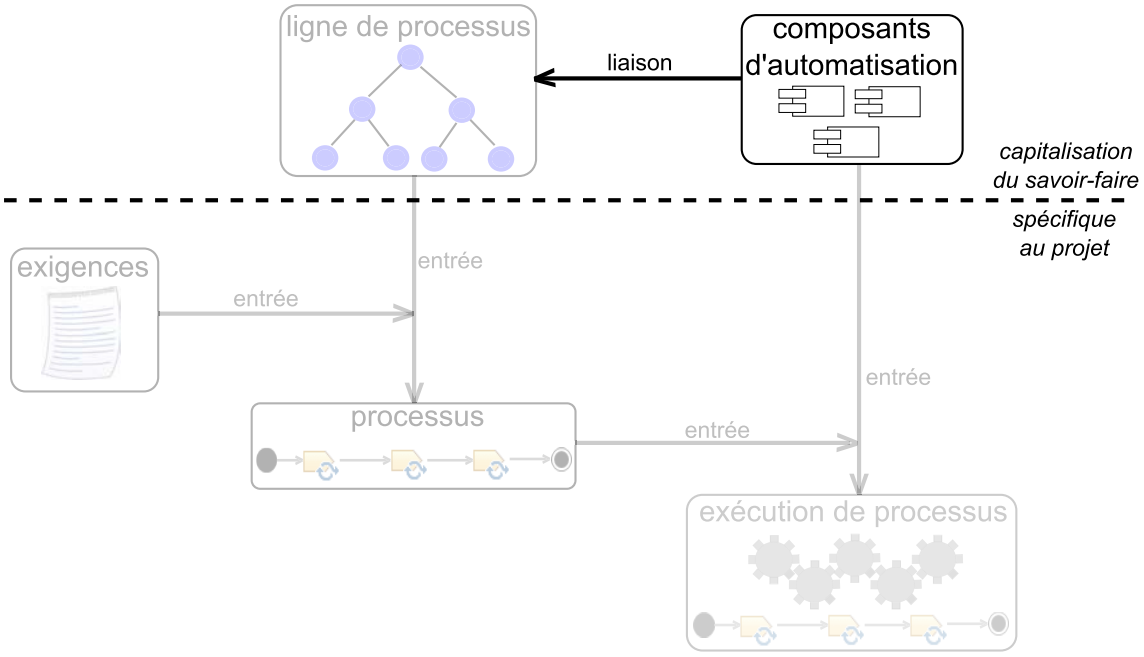


FIGURE 6.3 – Partie de l'approche supportée par les modeleurs de CA abstraits et de liaisons et le *framework* support à l'implémentation des CA

thèse qui est supportée par ces composants.

Le modeleur de liaisons permet de lier des CA abstraits (spécifiés à l'aide du modeleur de CA abstraits) aux unités de travail d'une ligne de processus qu'ils contribuent à automatiser. Le modeleur de liaisons permet également de lier les CA abstraits à leur implémentation. Lorsque plusieurs CA contribuent à l'automatisation d'une même unité de travail, ce modeleur doit permettre de spécifier l'ordre dans lequel ces CA doivent s'exécuter. Le modeleur de liaisons doit également permettre de spécifier si un CA contribue à automatiser l'initialisation, l'exécution ou la finalisation d'une unité de travail.

Le *framework* support à l'implémentation des CA permet premièrement aux CA d'accéder à des *informations contextuelles* nécessaires à leur exécution. Pour s'exécuter, les CA ont en effet besoin d'avoir accès à des informations spécifiques à leur contexte d'utilisation, c'est-à-dire à des informations contextuelles. Par exemple, un CA qui met du code source sous contrôle de version doit avoir accès à l'URL du dépôt distant sur lequel partager ce code source. Selon le langage de modélisation utilisé pour définir la ligne de processus, ces informations contextuelles peuvent, ou non, être capturées dans la ligne de processus (et donc dans le modèle de processus résolu). Par exemple, en SPEM 2.0, il n'est pas possible de capturer l'URL d'un dépôt distant. En effet, même si cette information peut, par exemple, être capturée en utilisant une recommandation (`Guidance`), elle ne serait pas typée comme étant une information relative à l'URL d'un dépôt distant et ne pourrait donc pas être traitée comme telle par un outil recherchant des informations contextuelles. Il est donc nécessaire d'avoir un mécanisme permettant aux CA d'accéder aux informations contextuelles, qu'elles soient ou non capturées dans le modèle de processus résolu.

Le *framework* support à l'implémentation des CA permet également d'exécuter les CA de manière générique. En effet, l'interpréteur de processus doit pouvoir être réutilisé indépendamment des CA. Or, il n'est pas possible de déterminer à l'avance un ensemble fini de CA puisque i) il n'est pas possible de prévoir toutes les évolutions qui peuvent être apportées à une famille de processus et ii) il n'est pas possible d'avoir une vision exhaustive de toutes les familles de processus existantes. Il est donc nécessaire de pouvoir, au fur et à mesure de l'exécution de processus, lancer l'exécution des CA de manière générique. Par ceci nous entendons plus précisément que le *framework* doit permettre de lancer automatiquement l'exécution des CA, quels qu'ils soient.

6.2.3 Dérivation d'un processus en fonction des exigences d'un projet

L'assistant à la résolution de la variabilité et le moteur de dérivation CVL supportent la dérivation d'un processus de la ligne de processus en fonction des exigences d'un projet, comme illustré par la figure 6.4. L'assistant à la résolution de la variabilité offre un support à la résolution de la variabilité, étape pouvant s'avérer fastidieuse et source d'erreurs. En effet, dans le VAM, la non séparation des préoccupations entre les spécifications de variabilité nécessitant une résolution manuelle et celles dont la résolution est dérivée (c'est-à-dire calculée automatiquement en fonction de la résolution d'autres spécifications de variabilité) peut compliquer la résolution de la variabilité

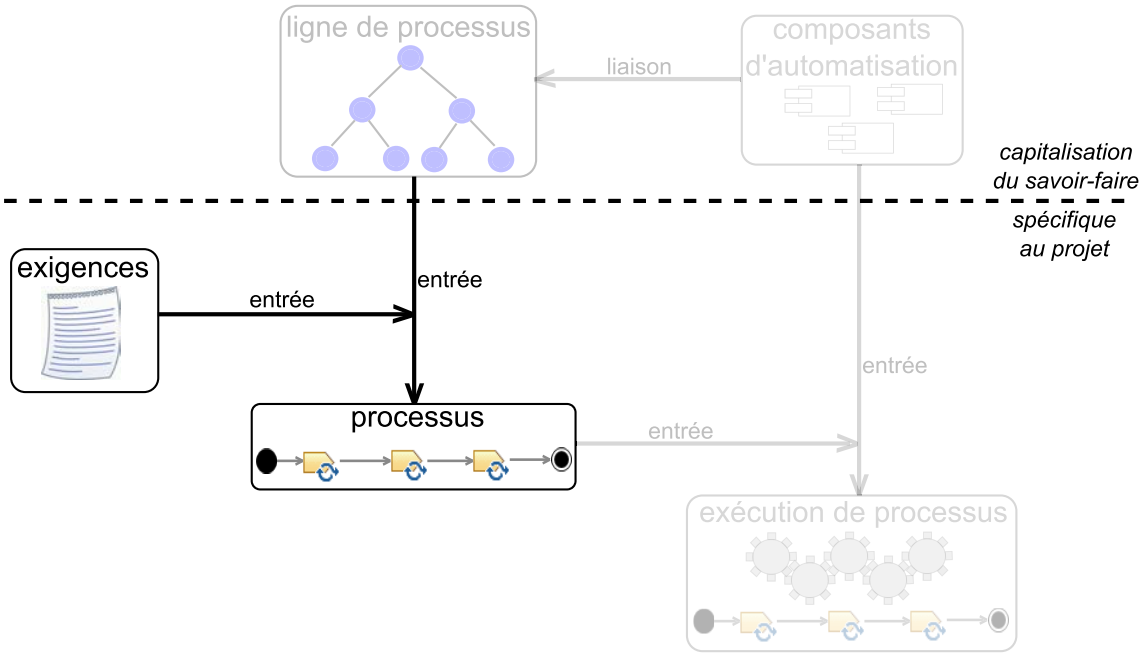


FIGURE 6.4 – Partie de l'approche supportée par l'assistant à la résolution de la variabilité et le moteur de dérivation CVL

pour un humain. De plus, plus le nombre de contraintes entre des spécifications de variabilité du VAM sera important, plus il sera difficile pour un humain de les prendre en compte. D'autre part, un humain peut toujours faire des erreurs conduisant au non-respect de la spécification de CVL. L'assistant à la résolution de la variabilité apporte donc un support i) en mettant en œuvre la séparation des préoccupations entre les spécifications de variabilité nécessitant une résolution manuelle et celles dont la résolution est dérivée, ii) en forçant le respect des contraintes entre les différentes spécifications de variabilité, et iii) en assurant le respect de la spécification de CVL.

Le moteur de dérivation CVL, quant à lui, dérive un processus de la ligne de processus conformément à la spécification de CVL. Afin de créer le modèle de processus résolu, il applique donc sur le modèle de processus de base les points de variation dont les spécifications de variabilité ont été résolues positivement.

6.2.4 Automatisation de l'exécution des processus

L'interpréteur de processus supporte l'automatisation de l'exécution des processus, comme illustré par la figure 6.5. Il exécute un processus résolu en fonction de la sémantique du langage de modélisation de processus utilisé. De plus, pour chaque unité de travail de ce processus, il lance, s'il y a lieu, l'exécution des CA associés. D'autre part, comme la réalisation de certaines unités de travail est manuelle [Ben07], l'interpréteur de processus informe l'acteur courant du processus lorsqu'une intervention manuelle est requise, afin d'assurer le bon déroulement de l'exécution. Enfin, comme

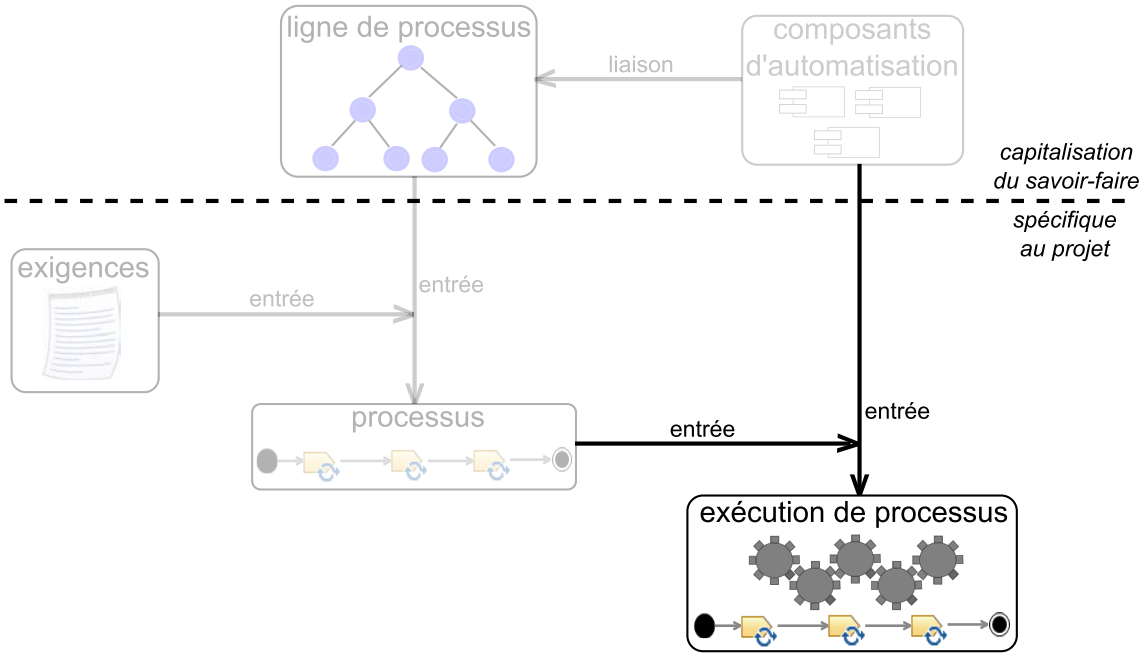


FIGURE 6.5 – Partie de l'approche supportée par l'interpréteur de processus

le processus à exécuter peut contenir de la variabilité non encore résolue, l'interpréteur de processus permet de résoudre cette variabilité au moment de l'exécution.

6.3 Implémentation

Les composants de T4VASP sont implémentés en Java, sous la forme de plug-ins Eclipse. Nous détaillons dans la suite de cette section comment nous les avons implémentés.

6.3.1 Les modeleurs de VAM et de VRM

Les modeleurs de VAM et de VRM correspondent chacun à un éditeur arborescent généré par défaut avec EMF, sur la base de fichiers Ecore capturant les parties du métamodèle de CVL relatives au VAM et au VRM. Les figures 6.6 et 6.7 illustrent respectivement des extraits du VAM et du VRM de l'exemple illustratif de famille de processus de métamodélisation, édités avec les modeleurs générés. L'extrait de VAM de la figure 6.6 comporte un choix sans enfant (*Choice checker*), un choix avec enfants (*Choice version control system*), ainsi qu'un choix dont la résolution est dérivée (*Choice parallelization*). Les enfants du choix *version control system* sont les choix *SVN* et *Git*. Ils sont mutuellement exclusifs, comme indiqué par la multiplicité de groupe valant 1 (*Multiplicity Interval 1*), élément spécifiant qu'un seul choix enfant peut être sélectionné.

L'extrait de VRM de la figure 6.7 contient deux points de variation, qui sont des

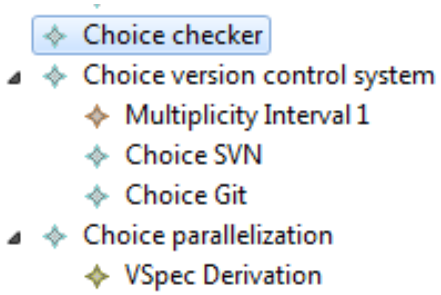


FIGURE 6.6 – Extrait du VAM de l'exemple illustratif de la famille de processus de modélisation, édité avec le modeleur de VAM

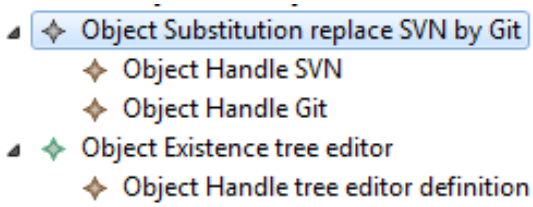


FIGURE 6.7 – Extrait du VRM de l'exemple illustratif de la famille de processus de modélisation, édité avec le modeleur de VRM

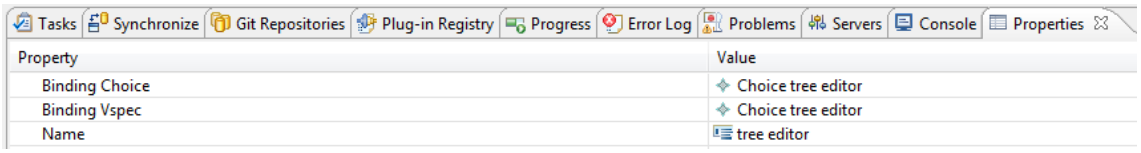


FIGURE 6.8 – Propriétés de l'existence d'objet *tree editor* de la figure 6.7

opérations à appliquer au modèle de base pour en dériver un modèle résolu (cf. section 2.3.2) : une substitution d'objet (*Object Substitution replace SVN by Git*) et une existence d'objet (*Object Existence tree editor*). Chacun de ces points de variation impacte des objets et des liens (*Object Handle SVN*, *Object Handle Git*, *Object Handle tree editor definition*).

Les modeleurs de VAM et de VRM ne font pas apparaître toutes les propriétés de chaque élément de modèle dans la vue arborescente. Par exemple, l'extrait de VRM de la figure 6.7 ne fait pas apparaître la spécification de variabilité associée à chaque point de variation. Les modeleurs de VAM et de VRM offrent donc en plus une vue exhaustive des propriétés de chaque élément de modèle, à travers laquelle elles peuvent être éditées. La figure 6.8 illustre les propriétés de l'existence d'objet *tree editor* de la figure 6.7. Cette figure permet de constater que la variabilité de spécification associée à l'existence d'objet *tree editor* est le choix *tree editor* (les propriétés *Binding Choice* et *Binding Vspec* ont pour valeur *Choice tree editor*).

6.3.2 Les modeleurs de CA abstraits et de liaisons

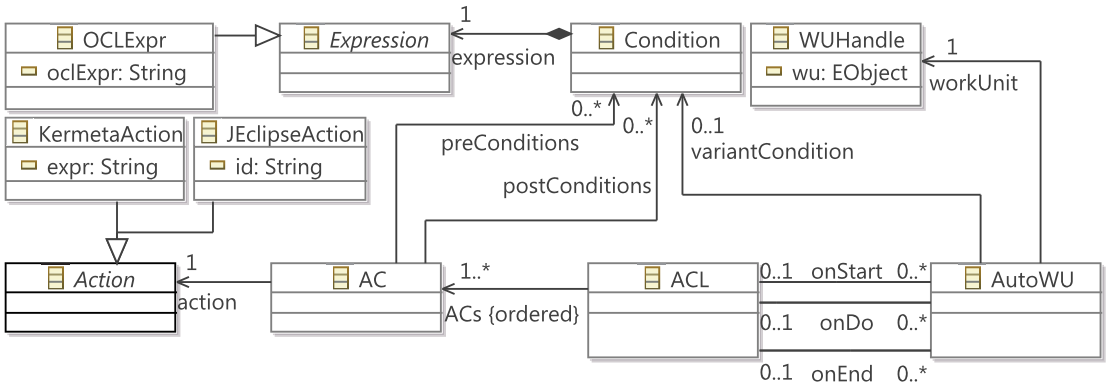


FIGURE 6.9 – Métamodèle de CA abstraits et de liaisons

Dans cette thèse, les modèles de CA abstraits et de liaisons sont définis dans un même modèle, à l'aide d'un modeleur commun. Nous appelons ce dernier le *modeleur de CA abstraits et de liaisons*. Afin de développer ce modeleur, nous avons utilisé EMF pour définir son métamodèle sous-jacent, le *métamodèle de CA abstraits et de liaisons* (cf. figure 6.9), et pour générer l'éditeur arborescent correspondant à ce métamodèle, c'est-à-dire le modeleur de CA abstraits et de liaisons en lui-même.

Dans le métamodèle de CA abstraits et de liaisons, une liste de CA (`ACL`, pour *Automation Component List*) automatise une unité de travail (`AutoWU`). L'intérêt de la classe `ACL` est de pouvoir réutiliser une liste d'AC. Les références `onStart`, `onDo` et `onEnd` entre une liste de CA et une unité de travail automatisée spécifient respectivement si la liste de CA automatise l'initialisation, l'execution ou la finalisation de l'unité de travail. Une liste de CA définit un ensemble de CA abstraits (`AC`, pour *Automation Component*). Une unité de travail automatisée référence une unité de travail de la ligne de processus via un pointeur d'unité de travail (`WUHandle`) et son attribut `MOFRef`, qui correspond à l'URI de l'unité de travail dans la ligne de processus. Si une unité de travail varie, une condition de variante (référence `variantCondition`) spécifie de quelle variante il s'agit. Des pré-conditions (référence `preConditions`) et post-conditions (référence `PostCond`) spécifient respectivement le contexte requis pour exécuter un CA et les résultats attendus de l'exécution d'un CA. La condition de variante ainsi que les pré et post conditions sont des conditions (`Condition`) exprimées à l'aide d'une expression OCL [OMG10] (`OCLExpr`).

La figure 6.10 illustre un extrait du modèle de CA abstraits et de liaisons de l'exemple illustratif, représenté sous forme de diagramme d'objets et non édité avec le modeleur de CA abstraits et de liaisons. Il comprend deux unités de travail automatisées (*NewInterpGit* et *NewInterpSVN*), qui sont deux variantes de la tâche de définition

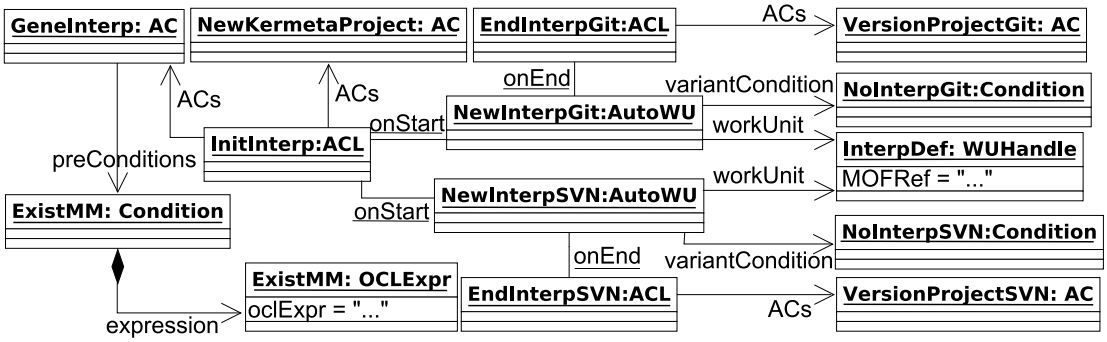


FIGURE 6.10 – Diagramme d'objets représentant un extrait du modèle de CA abstraits et de liaisons de l'exemple illustratif

d'un interpréteur (référencée par le pointeur d'unité de travail *InterpDef*). La condition de variante *NoInterpGit* (*NoInterpSVN*) spécifie que l'unité de travail automatisée *NewInterpGit* (*NewInterpSVN*) correspond à la tâche de définition de l'interpréteur lors de la première itération du processus et avec Git (SVN) comme SCV (Système de Contrôle de Version). La liste de CA *InitInterp* initialise cette tâche en créant un projet Kermeta (CA *NewKermetaProject*) et en générant le patron de conception interpréteur pour chaque métaclasse du métamodèle (CA *GeneInterp*). La finalisation de cette tâche consiste à mettre le code de l'interpréteur sous contrôle de version. La liste de CA *EndInterpGit* (*EndInterpSVN*) réalise cette mise sous contrôle de version via le CA *VersionProjectGit* (respectivement *VersionProjectSVN*) quand le SCV est Git (respectivement SVN). La pré-condition *ExistMM* spécifie que le CA *GeneInterp* requière l'existence du métamodèle pour son exécution.

Une action (`Action`) permet de lier un CA abstrait à son implémentation. Le métamodèle de CA abstraits et de liaisons définit différents types d'actions : action Kermeta (`KermetaAction`) et action Java Eclipse (`JEclipseAction`). Ils définissent la technologie utilisée pour implémenter les CA. Il est bien sûr possible d'ajouter de nouveaux types d'actions (Shell, Groovy,...) en fonction des besoins pour un cas d'application. L'implémentation d'une action Java Eclipse (respectivement d'une action Kermeta) est écrite dans un plug-in Eclipse (respectivement dans l'expression, identifiée par l'at-

tribut `expr`, de cette action Kermeta). C'est l'identifiant de l'action Java Eclipse qui permet de faire le lien avec le plug-in Eclipse implémentant cette action, comme nous allons le voir dans la section suivante.

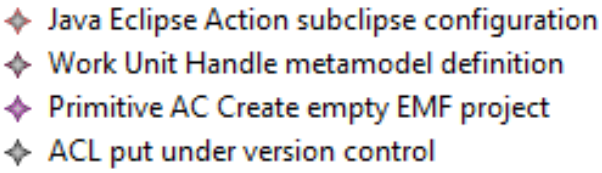


FIGURE 6.11 – Extrait du modèle de CA abstraits et de liaisons de l'exemple illustratif de la famille de processus de modélisation, édité avec le modeleur de CA abstraits et de liaisons

La figure 6.11 illustre un extrait du modèle de CA abstraits et de liaisons de l'exemple illustratif de famille de processus de métamodélisation, édité avec le modeleur de CA abstraits et de liaisons. À noter également que cet exrait est différent de celui de la figure 6.10. Cet extrait contient quatre éléments de modèle : une action Java Eclipse (*Java Eclipse Action subclipse configuration*), un pointeur d'unité de travail (*Work Unit Handle metamodel definition*), un CA (*Primitive AC Create empty EMF project*) et une liste de CA (*ACL put under version control*).

6.3.3 Le *framework* support à l'implémentation des CA

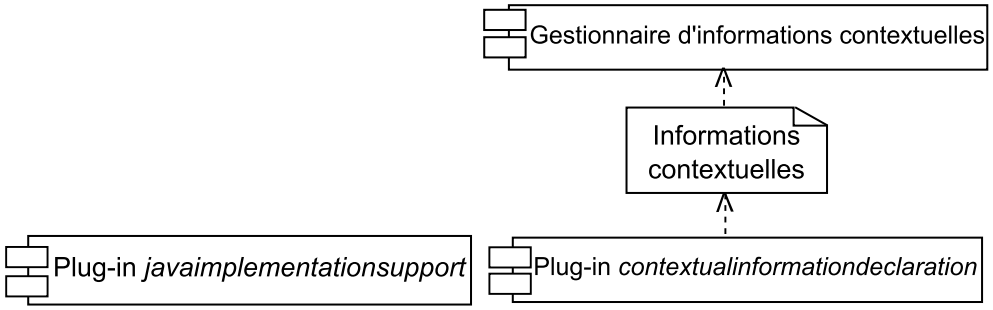


FIGURE 6.12 – Composants du framework

Comme illustré par la figure 6.12, le *framework* se compose :
- du plug-in *javaimplementationsupport*, qui permet de faire le lien entre un CA et le plug-in Eclipse l'implémentant et qui permet également d'exécuter de manière générique ce plug-in Eclipse,
- du plug-in *contextualinformationdeclaration*, qui permet aux CA de déclarer les informations contextuelles dont ils ont besoin pour s'exécuter,
- d'un gestionnaire d'informations contextuelles, dont le but est de trouver la valeur des informations contextuelles.

Nous détaillons ces trois composants dans la suite de cette section.

6.3.3.1 Liaison entre un CA et le plug-in Eclipse l'implémentant

Le plug-in *javaimplementationsupport* permet de faire le lien entre un CA spécifié dans le modèle de CA abstraits et de liaisons et le plug-in Eclipse l'implémentant. Ce plug-in fournit en effet un point d'extension, appelé *activityautomationregistry*, auprès duquel doit s'enregistrer un plug-in Eclipse implémentant un CA. Au moment de cet enregistrement, le lien entre un plug-in Eclipse et un CA est réalisé en spécifiant l'identifiant de l'action Java Eclipse correspondant à ce CA. La figure 6.13 illustre un exemple de déclaration de l'action Java Eclipse implémentée par un plug-in. Sur la partie droite de cette figure se trouvent les informations à renseigner au moment de l'enregistrement auprès du point d'extension *activityautomationregistry*. L'action Java Eclipse implémentée est déclarée en mettant son identifiant dans le champ *action*.

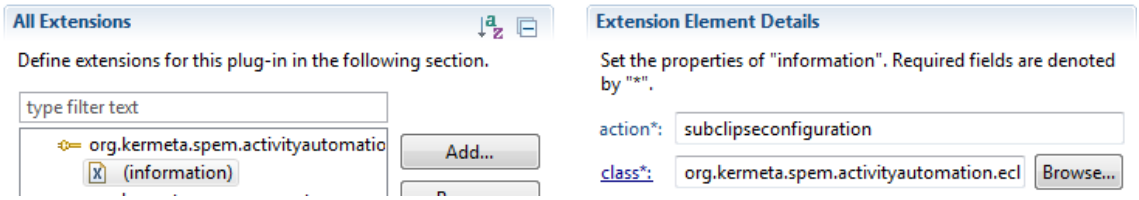


FIGURE 6.13 – Enregistrement auprès du point d'extension *activityautomationregistry*

6.3.3.2 Exécution générique des plug-in Eclipse implémentant les CA

Afin de pouvoir lancer de manière générique l'exécution des CA implémentés en Java avec des plug-ins Eclipse, le plug-in *javaimplementationsupport* fournit une interface, appelée *ActivityAutomation*, qui doit être implémentée par chaque plug-in. L'implémentation de cette interface consiste à implémenter une méthode *run()*. C'est cette méthode qui est appelée pour lancer l'exécution d'un CA. Le corps de cette méthode doit donc contenir l'implémentation d'un CA. Pour pouvoir appeler la méthode *run()* de cette interface, il est nécessaire, pour des raisons techniques, de préciser quelle est la classe (d'un plug-in Eclipse correspondant à l'implémentation d'un CA) qui implémente cette méthode. Encore une fois, cette information est fournie au moment de l'enregistrement auprès du point d'extension *activityautomationregistry* du plug-in correspondant à l'implémentation d'un CA. La figure 6.13 illustre également un exemple de déclaration d'une classe implémentant l'interface `ActivityAutomation`. Le nom de cette classe est spécifié dans le champ *class* dans la liste des informations à renseigner au moment de l'enregistrement auprès du point d'extension *activityautomationregistry*.

Toujours afin de pouvoir lancer les CA de manière générique, l'interpréteur de processus doit pouvoir les instancier de manière générique. Chaque classe implémentant l'interface `ActivityAutomation` doit donc avoir un constructeur Java par défaut.

6.3.3.3 Gestion de l'accès aux informations contextuelles

Chaque CA déclare la liste des informations contextuelles nécessaires à son exécution en s'enregistrant auprès du point d'extension *parameterregistry*, fournit par le plug-in *contextualinformationdeclaration*. Le gestionnaire d'informations contextuelles se charge d'affecter des valeurs aux informations contextuelles déclarées par les CA. Comme illustré par la figure 6.14, si une information contextuelle peut être capturée par le modèle du processus en cours d'exécution, alors le gestionnaire récupère sa valeur dans ce modèle. Sinon, il la récupère dans un *modèle de contexte*, qui contient un ensemble de clés/valeurs associant une information contextuelle à sa valeur. L'interpréteur de processus crée un modèle de contexte pour chaque exécution de processus. Si une information contextuelle n'est dans aucun modèle, alors le gestionnaire la demande à l'acteur courant du processus et la sauve dans le modèle de contexte afin qu'elle puisse être réutilisée. Le gestionnaire n'a pas besoin de sauver une information contextuelle dans le modèle du processus en cours d'exécution. En effet, si une information contextuelle pouvant être capturée par le modèle de processus ne l'est pas, alors il s'agit d'un cas de variabilité non résolue qui est traité par l'interpréteur de processus. Au moment de la déclaration des informations contextuelles nécessaires à l'exécution d'un CA, il est possible de spécifier qu'une information est confidentielle. Dans ce cas, cette information n'apparait pas en clair si elle est saisie par l'acteur courant.

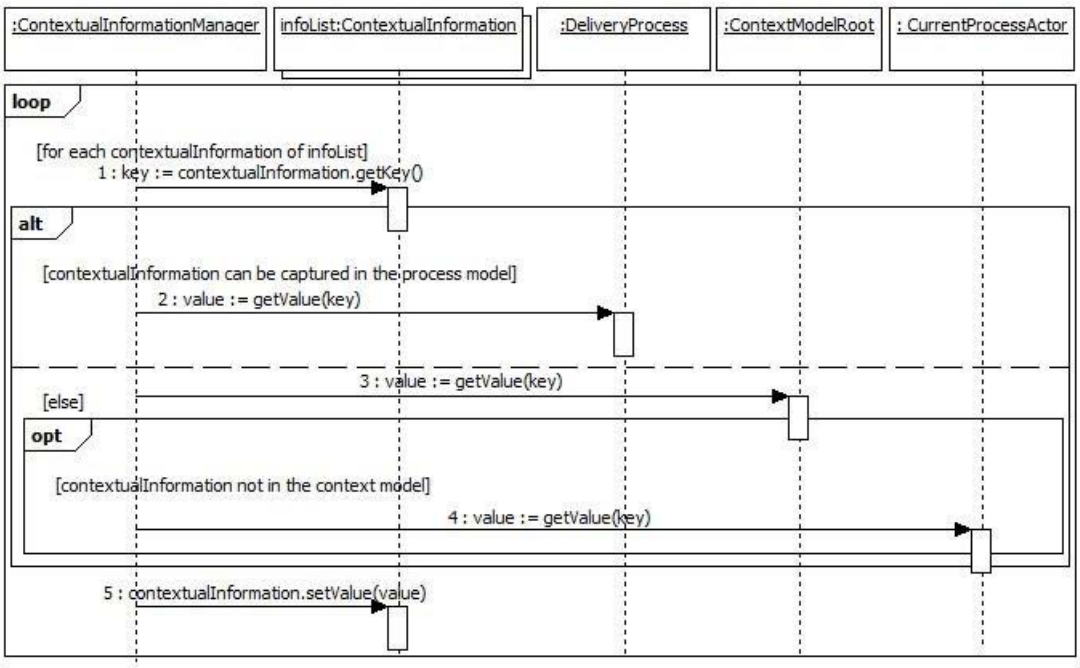


FIGURE 6.14 – Comportement du gestionnaire d'informations contextuelles

Lorsqu'un CA est réutilisé plusieurs fois dans un même processus, mais avec des

artéfacts différents, alors les informations contextuelles liées à ces artéfacts, qui ne peuvent pas être capturées dans le modèle de processus, sont à demander à l'acteur courant, tant qu'elles n'ont pas été écrites dans le modèle de contexte. Par exemple, un CA qui crée un projet Kermeta a besoin pour s'exécuter du nom du projet Kermeta à créer. Or, ce nom sera différent pour chaque projet à créer. Le gestionnaire d'informations contextuelles devra donc demander à chaque fois le nom du projet à créer à l'acteur courant. Cependant, lorsqu'une information contextuelle spécifique à un artéfact est réutilisée par un CA, il faut pouvoir la différencier d'une information contextuelle capturant le même type d'information mais spécifique à un autre artéfact. Par exemple, il faut pouvoir différencier les noms de deux projets Kermeta. Ce problème ne se pose pas avec les informations contextuelles pouvant être capturées dans le modèle de processus. Il est en effet dans ce cas possible de les différencier car elles sont liées à des artéfacts différents.

Des CA différents sont utilisés pour demander des informations contextuelles qui doivent être différenciées. Par exemple, dans le cas de la création de projets Kermeta, chaque projet a son propre CA. Chaque CA déclare l'information contextuelle correspondant au nom du projet à créer avec une clé spécifique, afin que celle-ci puisse être différenciée des autres clés dans le modèle de contexte. Afin d'éviter les redondances entre les CA, les traitements métiers qui leur sont communs (par exemple la création d'un projet) sont factorisés dans une même fonction, prenant en paramètres les informations contextuelles spécifiques à chaque CA.

Cette manière de gérer les informations contextuelles ne fonctionne que si tous les cas d'utilisation d'un CA sont connus au moment de la définition des CA. Cela n'est pas toujours le cas. Par exemple, lors d'un processus de développement Java, plusieurs projets Java peuvent être créés mais leur nombre n'est souvent pas connu à l'avance. Il n'est donc pas possible de créer autant de CA mettant un projet Java sous contrôle de version que de projets Java. Dans ce cas, un seul CA est créé pour automatiser une TMR particulière. Il demande à l'acteur courant les informations contextuelles spécifiques à des artéfacts et non encore sauvées dans le modèle de contexte. Il écrase l'ancienne valeur de l'information contextuelle dans le modèle de contexte si elle existe, et sinon il crée un nouveau couple clé/valeur.

Cette deuxième manière de gérer les informations contextuelles ne fonctionne que si l'ancienne valeur d'une information contextuelle n'est plus utilisée. Dans le cas contraire, une information contextuelle est toujours demandée à l'acteur courant sans jamais la sauver dans le modèle de contexte. Cette dernière manière de gérer les informations contextuelles implique de solliciter plusieurs fois l'acteur courant pour lui demander la même chose, ce qui est source d'erreurs. Nous ne la réservons donc que pour les cas où les deux premières manières de procéder ne sont pas adéquates.

Le gestionnaire d'informations contextuelles doit être en mesure de déterminer si i) une information n'est à demander à l'acteur courant que si elle n'est pas déjà dans le modèle de contexte, ii) une information est à demander à l'acteur courant et son ancienne valeur doit être écrasée dans le modèle de contexte, ou iii) une information est demandée à l'acteur courant sans sauvegarde dans le modèle de contexte. À cette fin, les CA, lorsqu'ils déclarent les informations contextuelles nécessaires à leur exécution,

spécifient comment chaque information doit être traitée.

Afin que l'acteur courant puisse se concentrer sur la réalisation d'autres tâches pendant que les CA s'exécutent, il est nécessaire qu'il n'ait pas à donner des informations contextuelles aux CA au fur et à mesure de leur exécution. Aussi, le gestionnaire récupère les valeurs de toutes les informations contextuelles nécessaires à l'exécution d'une liste de CA au début de l'exécution de cette liste.

Le principal intérêt du gestionnaire d'informations contextuelles est qu'il découple l'accès aux informations contextuelles de l'implémentation des CA. Cela permet de rendre les CA plus réutilisables. En effet, l'accès aux informations contextuelles (et donc le gestionnaire d'informations contextuelles) dépend du langage de modélisation de processus utilisé, puisque selon ce langage une information contextuelle est récupérée dans le modèle du processus en cours d'exécution ou dans le modèle de contexte. Gérer l'accès aux informations contextuelles directement dans l'implémentation des CA aurait donc rendu ces CA dépendant du langage de modélisation de processus utilisé.

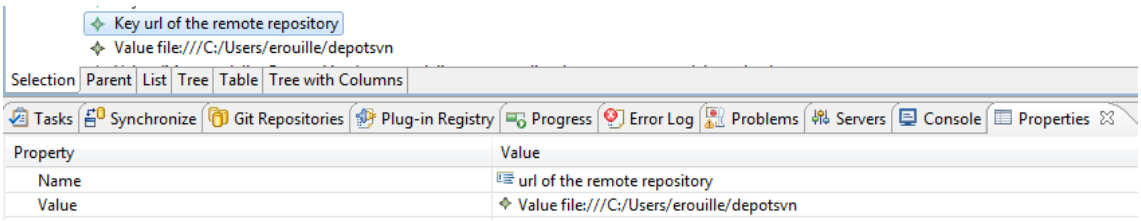


FIGURE 6.15 – Extrait du modèle de contexte d'un processus de métamodélisation avec contrôle de version

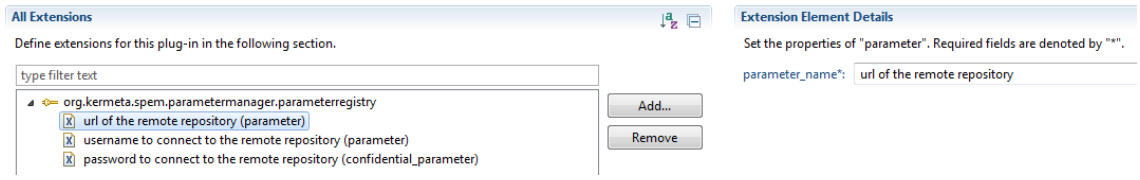


FIGURE 6.16 – Exemple de déclaration par un CA d'informations contextuelles

La figure 6.15 illustre un extrait du modèle de contexte d'un processus de métamodélisation avec contrôle de version. Ce modèle de contexte contient la clé *url of the remote repository*, qui correspond à l'URL du dépôt distant sur lequel partager des ressources. À cette clé est associée la valeur *file :///C :/Users/erouille/depotsvn*.

La figure 6.16 illustre un exemple de déclaration par un CA des informations contextuelles nécessaires à son exécution. Il s'agit ici d'un CA qui connecte l'espace de travail Eclipse d'un acteur courant du processus à un nouveau dépôt SVN distant. Ce CA a besoin pour s'exécuter de trois informations contextuelles : l'URL du dépôt SVN distant (*url of the remote repository*), le nom d'utilisateur (*username to connect to the remote repository*) et le mot de passe (*password to connect to the remote repository*) de

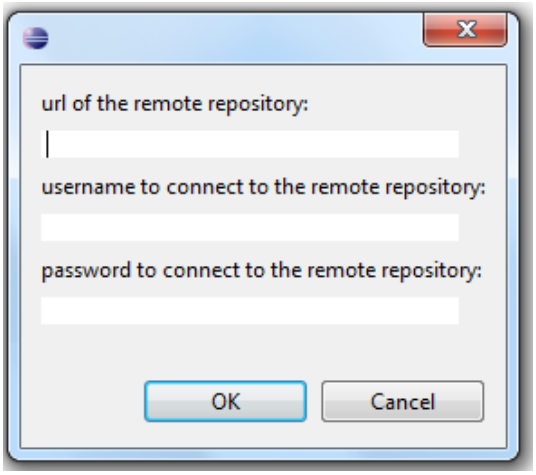


FIGURE 6.17 – Exemple de demande d'informations contextuelles à l'acteur courant du processus

l'acteur courant. Cette dernière information est ici déclarée comme confidentielle. La figure 6.17 illustre la demande de ces informations contextuelles à l'acteur courant d'un processus de métamodélisation de l'exemple illustratif. Cette demande a lieu au moment de la finalisation de la définition d'un interpréteur. Pour rappel, la finalisation de la définition d'un interpréteur consiste à mettre cet interpréteur sous contrôle de version. Plus précisément, la mise sous contrôle de version d'un interpréteur consiste à connecter l'espace de travail Eclipse du développeur de l'interpréteur à un dépôt SVN distant, puis à exporter sur ce dépôt le projet Kermeta contenant le code source de l'interpréteur. Les informations contextuelles demandées ne peuvent pas être capturées dans un modèle de processus SPEM et n'ont pas déjà été sauvées dans le modèle de contexte à ce stade de l'exécution du processus.

6.3.4 L'assistant à la résolution de la variabilité

L'assistant à la résolution de la variabilité que nous avons développé parcourt un VAM et pour chaque spécification de variabilité de ce VAM dont la résolution n'est pas dérivée, il demande à l'utilisateur de la ligne de processus de résoudre la variabilité via une interface. La manière de résoudre la variabilité qui est proposée dans l'interface dépend du type de spécification de variabilité, afin d'assurer le respect de la spécification de CVL. Par exemple, pour faire un choix, l'utilisateur de la ligne de processus ne peut que donner une réponse booléenne en sélectionnant ou non une case à cocher. De plus, la résolution de certaines spécifications de variabilité peut être forcée afin d'assurer le respect des contraintes entre les spécifications de variabilité du VAM. Par exemple, si un choix est obligatoire il est automatiquement sélectionné et l'utilisateur de la ligne de processus ne peut pas modifier cette résolution. Si deux choix sont mutuellement exclusifs alors l'interface ne permet de sélectionner qu'un seul de ces deux choix. Si des choix peuvent être combinés, l'interface permet alors de

sélectionner plusieurs de ces choix. L'assistant à la résolution de la variabilité crée dans un RM les résolutions de spécifications de variabilité (`VSpecResolution`, section 2.3.2). L'utilisateur de la ligne de processus a également la possibilité de passer la résolution d'une spécification de variabilité, dans le cas où la variabilité ne peut pas être résolue tout de suite. Dans ce cas, les résolutions de spécification de variabilité associées ne sont pas créées dans le RM.

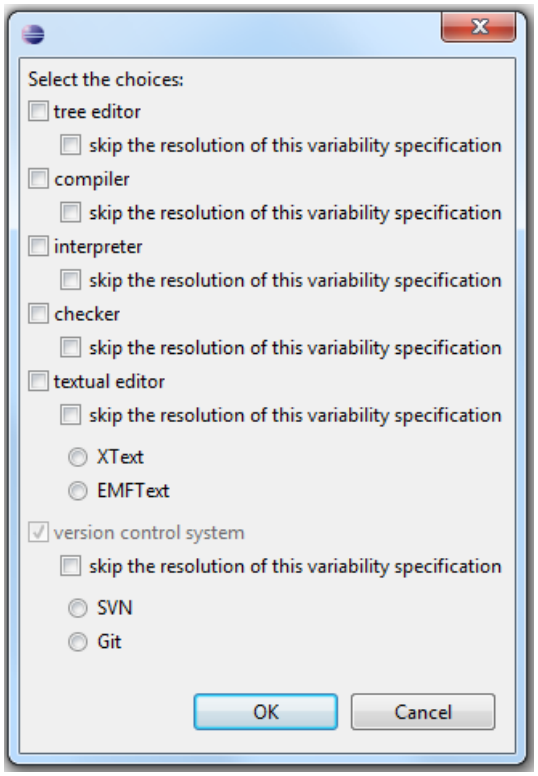


FIGURE 6.18 – Utilisation de l'assistant à la résolution de la variabilité avec l'exemple illustratif de famille de processus de métamodélisation

La figure 6.18 illustre l'utilisation de l'assistant à la résolution de la variabilité avec l'exemple illustratif de famille de processus de métamodélisation. Cet assistant permet à un utilisateur de la ligne de processus de résoudre la variabilité du VAM de la famille de processus de métamodélisation (cf. figure 4.9 pour un extrait de ce VAM) en résolvant les choix *tree editor*, *compiler*, *interpreter*, *checker*, *textual editor*, *XText*, *EMFText*, *SVN* et *Git*. Le choix *version control system* étant obligatoire, il est déjà sélectionné et il n'est pas possible de changer cette résolution. Les choix *tree editor*, *compiler*, *interpreter*, *checker* et *textual editor* n'étant pas mutuellement exclusifs, des cases à cocher sont utilisées pour les résoudre. Comme les choix *XText* et *EMFText* d'une part et *SVN* et *Git* d'autre part sont mutuellement exclusifs, des boutons radio sont utilisés pour les résoudre. Il est possible de passer la résolution de la variabilité d'un choix en cochant la case *skip the resolution of this variability specification* associée à ce choix.

6.3.5 Le moteur de dérivation CVL

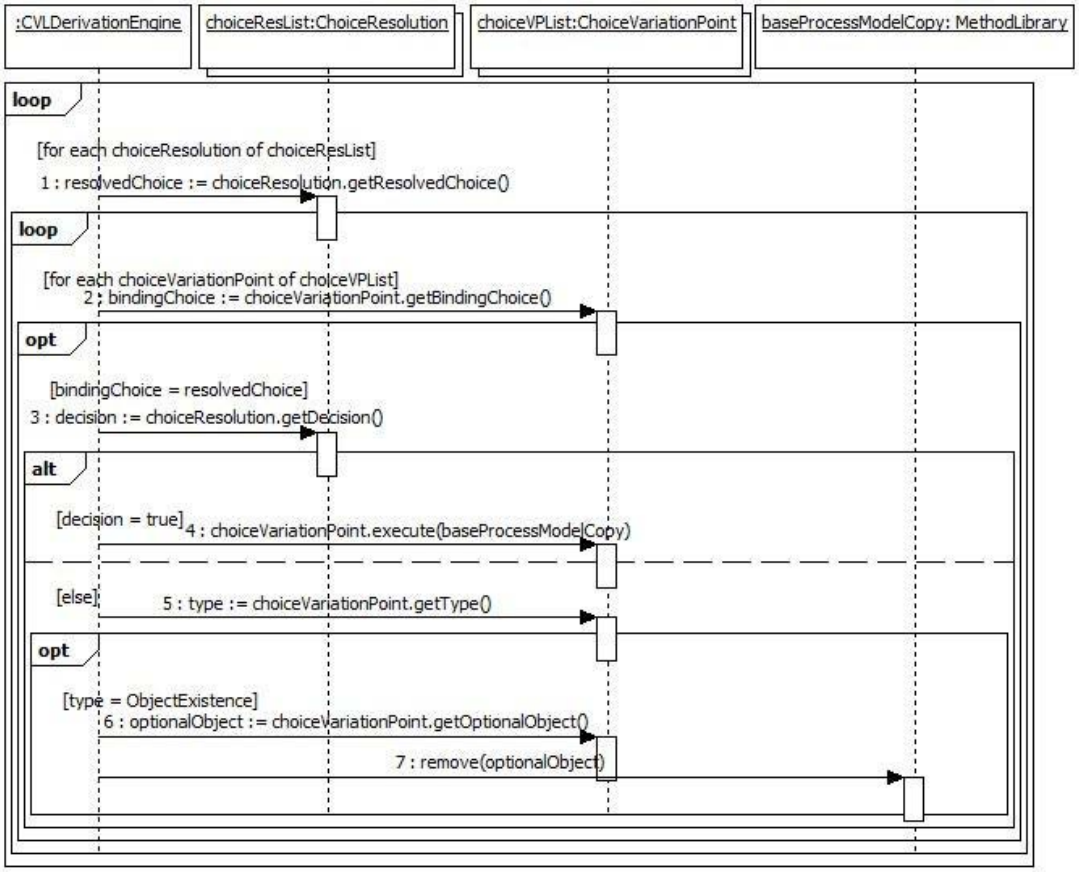


FIGURE 6.19 – Comportement du moteur de dérivation CVL

Nous décrivons ici le fonctionnement du moteur de dérivation CVL que nous avons implémenté dans le cadre de cette thèse. Comme illustré par la figure 6.19, pour chaque résolution de choix d'un RM, le moteur de dérivation CVL trouve le choix correspondant dans le VAM. Ensuite, le moteur de dérivation CVL trouve les points de variation qui sont associés à ce choix. Si le choix est sélectionné, ces points de variation sont exécutés conformément à la sémantique définie dans la spécification de CVL. Le cas échéant, les objets optionnels associés à des points de variation de type `ObjectExistence` sont supprimés. Les points de variation s'exécutent sur une copie du modèle de processus de base, afin de créer un modèle de processus résolu sans écraser le modèle de processus de base.

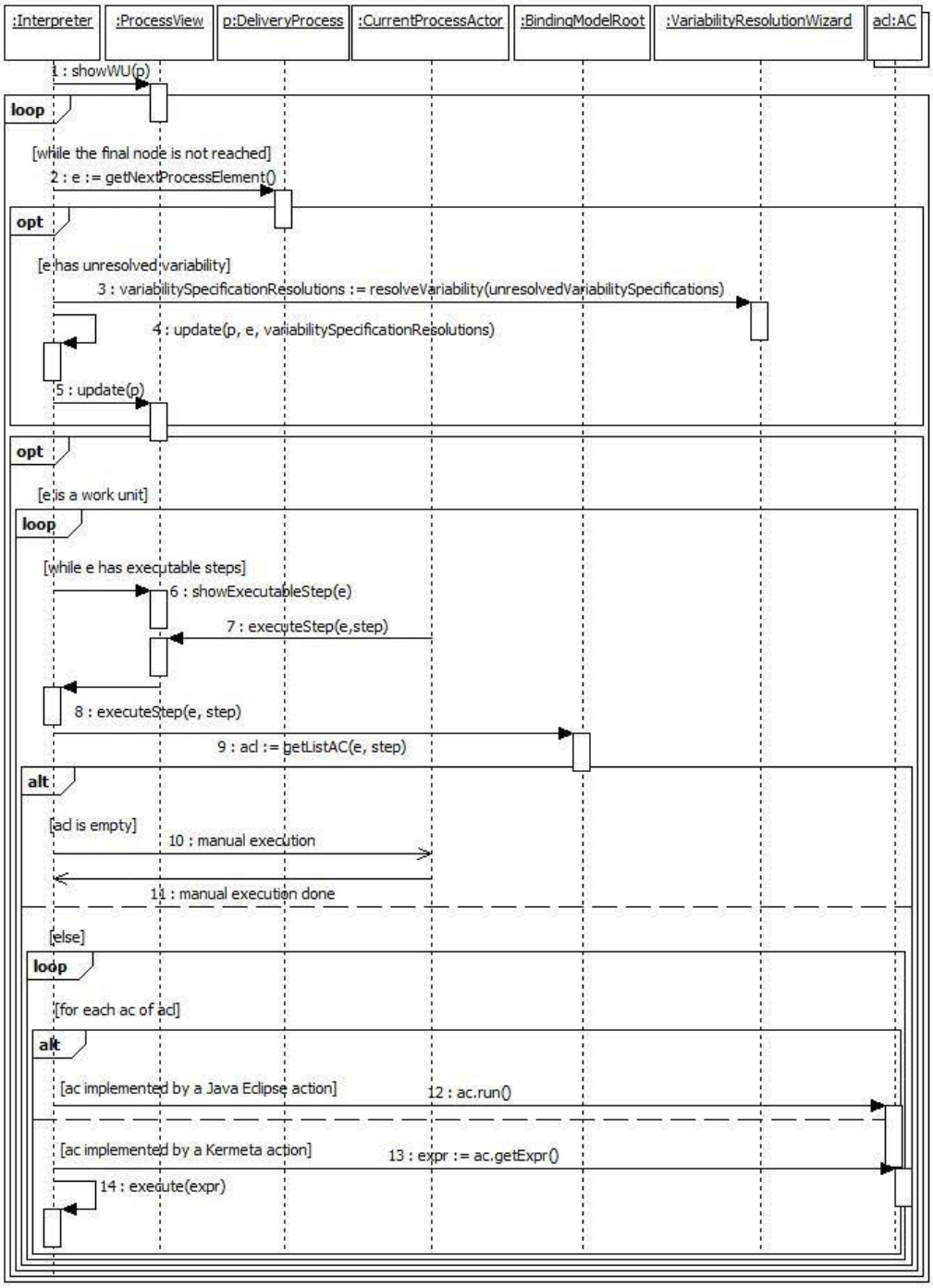


FIGURE 6.20 – Comportement de l'interpréteur de processus

6.3.6 L'interpréteur de processus

Nous détaillons maintenant l'interpréteur de processus développé dans le cadre de cette thèse. Son fonctionnement est illustré par la figure 6.20. Lors de son lancement, l'interpréteur de processus commence par afficher dans une vue spécifique, la *vue processus*, les différentes unités de travail qui composent le processus à exécuter. L'interpréteur de processus démarre ensuite l'exécution du processus, en fonction de la sémantique des diagrammes d'activités UML 2. Cependant, avant l'exécution d'un élément de processus, l'interpréteur de processus détermine si de la variabilité non résolue est associée à cet élément de processus. Si c'est le cas, l'interpréteur de processus appelle l'assistant à la résolution de la variabilité avec en paramètre les spécifications de variabilité non résolues associées à l'élément de processus courant, afin que l'acteur courant du processus résolve cette variabilité. Une fois la variabilité résolue, l'interpréteur met à jour le processus en cours d'exécution en fonction de la résolution de la variabilité, et met également à jour la vue processus. Pour mettre à jour le processus en cours d'exécution, l'interpréteur exécute sur ce processus les points de variation associés aux spécifications de variabilité qui viennent d'être résolues positivement.

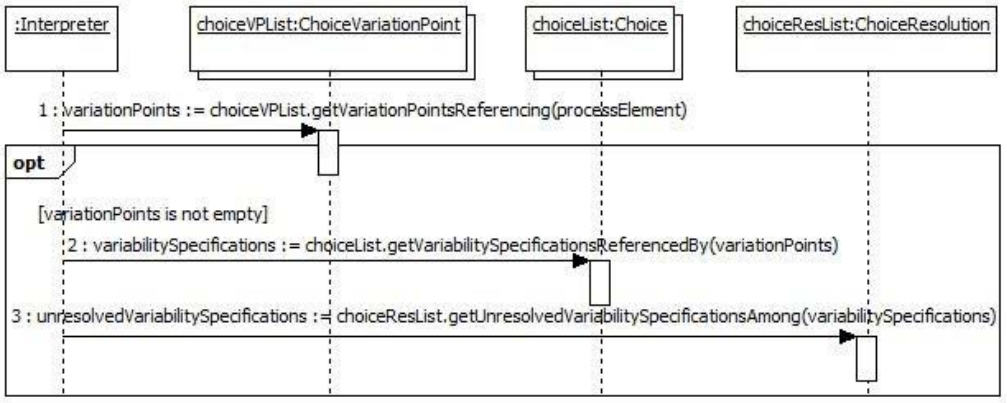


FIGURE 6.21 – Détection de variabilité non résolue

Pour déterminer si de la variabilité non résolue est associée à un élément de processus, l'interpréteur cherche dans le VRM si des points de variation référencent cet élément de processus. Si de tels points de variation existent, l'interpréteur vérifie dans le RM si les spécifications de variabilité associées à ces points de variation sont toutes résolues. Si tel n'est pas le cas, cela signifie que de la variabilité n'est pas résolue. La figure 6.21 illustre la détection de variabilité non résolue par l'interpréteur.

Si l'élément de processus à exécuter est une unité de travail, alors l'interpréteur affiche dans la vue processus l'étape (initialisation, exécution, finalisation) de cette unité de travail qui peut être lancée. L'ordre entre les étapes d'une unité de travail est respecté. C'est-à-dire qu'une étape peut être lancée seulement si les étapes précédentes ont été réalisées. Par exemple, la finalisation ne peut être lancée que si l'exécution a déjà

été réalisée. De plus, une initialisation ou une finalisation ne peut être lancée que si elle est automatisée, alors qu'une exécution peut être lancée qu'elle soit automatisée ou manuelle. Lorsque la finalisation d'une unité de travail est terminée (ou son exécution si elle n'a pas de finalisation), l'interpréteur de processus exécute les éléments de processus qui suivent cette unité de travail dans le processus. C'est l'acteur courant qui lance l'initialisation, l'exécution ou la finalisation d'une unité de travail. Cela lui permet de suivre l'exécution du processus. Ainsi, s'il a une tâche manuelle à réaliser pour participer à l'exécution du processus, il saura où l'exécution en est rendue et donc ce qui a déjà été fait et qu'il n'a pas à refaire.

L'initialisation ou la finalisation d'une unité de travail consiste à lancer automatiquement l'exécution de la liste de CA associée. L'exécution d'une unité de travail consiste à faire de même lorsque celle-ci est automatisée. Sinon, l'interpréteur informe l'acteur courant qu'il doit réaliser l'exécution manuellement et attend que ce soit fait avant de continuer l'exécution du processus. Lancer l'exécution d'une liste de CA consiste à lancer l'exécution de ses CA, dans leur ordre de définition. Pour lancer l'exécution d'un CA associé à une action Kermeta, l'interpréteur appelle une fonction qui évalue dynamiquement l'expression de cette action. Pour lancer l'exécution d'un CA associé à une action Java Eclipse, l'interpréteur trouve le plug-in qui est relié à cette action et appelle la méthode *run()* de la classe implémentant le CA.

Si une unité de travail est optionnelle, l'interpréteur de processus offre à l'acteur courant la possibilité de passer l'exécution de cette unité de travail. Attention, ici nous considérons une unité de travail qui est optionnelle dans un processus résolu, ce qui dénote donc de la variabilité au moment de l'exécution du processus (et non au moment de sa conception), tout comme les nœuds de décision par exemple.

L'interpréteur de processus capture également les traces d'exécution de processus, afin de savoir quelle est l'itération de processus en cours et par quels flots de contrôle une unité de travail est accédée ou quittée. L'exécution de certains CA dépend en effet de ce qui se passe avant ou après l'unité de travail qu'ils automatisent. Par exemple, dans le cas de la famille de processus de métamodélisation, le CA créant un nouveau projet Kermeta au début de la tâche de définition d'un interpréteur ne s'exécute que lors de la première itération du processus.

La figure 6.22 illustre la vue processus d'une variante de processus de métamodélisation comprenant les tâches de définition d'un métamodèle, d'un éditeur arborescent, d'un interpréteur, d'un compilateur et d'un vérificateur. Chacune de ces tâches a des boutons *on start*, *on do*, *done*, *on end* et *skip*. Ces boutons permettent de lancer l'initialisation d'une unité de travail (*on start*), de lancer l'exécution d'une unité de travail (*on do*), de signaler que l'exécution manuelle d'une unité de travail est terminée (*done*), de lancer la finalisation d'une unité de travail (*on end*) et de passer l'exécution d'une unité de travail lorsque celle-ci est optionnelle (*skip*). Sur la figure 6.22, l'exécution du processus vient juste de démarrer. Aussi, seule l'initialisation de la tâche de définition d'un métamodèle, ici automatisée, peut être lancée. En effet, sur la figure 6.22 seul le bouton *on start* lié à cette tâche est actif, les autres étant grisés.

FIGURE 6.22 – Vue processus d'une variante de processus de métamodélisation

6.4 Discussion

Nous énonçons dans cette partie des recommandations pour implémenter les CA et les relier à leurs contextes d'utilisation (section 6.4.1), puis nous abordons les extensions possibles à T4VASP (section 6.4.2).

6.4.1 Recommandations

Lorsque la ligne de processus spécifie qu'un outil peut être substitué par un autre, alors, dans l'implémentation des CA, les parties qui dépendent de cet outil doivent être découplées des parties qui n'en dépendent pas. Cela permet de pouvoir réutiliser ces dernières lorsque l'outil est remplacé.

Dans le modèle de CA abstraits et de liaisons, l'architecte doit spécifier pour quels séquencements et pour quelles itérations d'une unité de travail des CA sont appropriés, si les CA dépendent de ces séquencements et de ces itérations. En effet, la variabilité de ce qui se passe avant ou après une unité de travail (entre les différentes variantes d'un processus ou au sein d'une même variante de processus) peut impliquer de la variabilité au niveau des CA qui automatisent cette unité de travail. Des CA peuvent donc être requis, inutiles ou inutilisables en fonction de ce qui se passe avant ou

après une unité de travail. De plus, une unité de travail peut varier en fonction de ses itérations. Par exemple, la première itération d'une unité de travail peut consister à créer des artéfacts, tandis que les itérations suivantes consistent à modifier ces derniers. Cela peut impliquer des variations des CA qui automatisent cette unité de travail ainsi que des CA automatisant les unités de travail précédentes et suivantes.

6.4.2 Extensions

Actuellement, T4VASP fonctionne uniquement dans un environnement local. Or, plusieurs personnes peuvent participer à la réalisation d'un processus de développement logiciel, chacune avec son propre poste informatique. Il est donc important que l'automatisation de l'exécution d'un processus puisse se faire dans un environnement distribué. Une extension serait donc que T4VASP devienne une application client-serveur, et que l'exécution d'un processus soit centralisée sur la partie serveur, tandis que les automatisations des TMR s'exécuteraient sur les postes clients. Ainsi, plusieurs personnes pourraient partager l'exécution d'un même processus.

L'interpréteur de processus ne permet actuellement pas à l'acteur courant du processus de lancer l'initialisation ou la finalisation d'une unité de travail si ces initialisations et finalisations ne sont pas automatisées. En effet, comme SPEM 2.0 ne permet pas de définir d'initialisation et de finalisation à une unité de travail, l'interpréteur ne peut pas savoir si une unité de travail a une initialisation ou une finalisation, à moins que celle-ci soit automatisée car cela est dans ce cas spécifié dans le modèle de CA abstraits et de liaisons. Si une initialisation ou une finalisation n'est pas spécifiée elle peut être omise par l'acteur courant du processus. Lorsqu'une unité de travail a une initialisation ou une finalisation manuelle, la seule manière de le spécifier en SPEM 2.0 est de faire précéder (suivre) cette unité de travail par une autre unité de travail consistant en l'initialisation (la finalisation). Par exemple, si l'unité de travail *U1* a une initialisation manuelle, cela pourra apparaître dans un modèle de processus SPEM 2.0 en faisant précéder *U1* d'une unité de travail *U0* consistant à réaliser l'initialisation. Mais l'inconvénient de cette solution est que les initialisations et les finalisations ne sont pas fortement couplées aux unités de travail qu'elles initialisent ou finalisent. Il est donc possible pour un humain d'omettre l'initialisation ou la finalisation d'une unité de travail lors de sa réutilisation. Par exemple lors de la création d'un nouveau modèle de processus contenant l'unité de travail *U1* précédente, la personne en charge de la modélisation peut omettre de faire précéder *U1* de *U0*. Une extension à T4VASP serait donc de faire évoluer le métamodèle de CA abstraits et de liaisons de manière à pouvoir spécifier qu'une initialisation ou une finalisation est manuelle.

Dans le cas où une initialisation, exécution ou finalisation d'unité de travail est composée à la fois d'étapes automatisées et manuelles, l'outillage actuel impose de définir dans le modèle de processus les unités de travail à un niveau de granularité plus fin, de sorte qu'une initialisation, exécution ou finalisation soit complètement manuelle ou complètement automatisée. L'outillage actuel ne prend en effet pas en compte les initialisations, exécutions ou finalisations partiellement automatisées. Ainsi, si l'exécution d'une unité de travail *U1* consiste par exemple à réaliser trois actions, *A1*, *A2*

et *A3*, avec *A1* et *A3* automatisées et *A2* manuelle, alors dans le modèle de processus *U1* devra être décomposée en trois unités de travail, consistant respectivement en la réalisation des actions *A1*, *A2* et *A3*. Si les actions *A1* et *A3* peuvent être considérées comme les initialisations et finalisations respectives de *A2*, une autre solution serait de définir *A1* comme initialisation de *U1*, *A2* comme exécution de *U1* et *A3* comme finalisation. Cela conduit à des définitions de processus plus détaillées. Afin de satisfaire le cas où les utilisateurs d'un modèle de processus préfèrent une définition plus abstraite, une extension à T4VASP serait de faire évoluer le métamodèle de CA abstraits et de liaisons de manière à pouvoir spécifier des listes d'AC qui soient entrecoupées de tâches manuelles et ainsi pouvoir gérer les initialisations, exécutions ou finalisations partiellement automatisées.

6.5 Synthèse

Nous avons présenté dans ce chapitre T4VASP, l'outil support à l'approche proposée dans cette thèse. T4VASP se compose de 8 composants : un modeleur de VAM, un modeleur de VRM, un assistant à la résolution de la variabilité, un moteur de dérivation CVL, un modeleur de CA abstraits et de liaisons, un *framework* supportant l'implémentation des CA ainsi qu'un interpréteur de processus. T4VASP permet de définir une ligne de processus, de lier des CA à cette ligne de processus, de dériver automatiquement un processus de la ligne de processus en fonction des exigences d'un projet et d'automatiser l'exécution d'un processus. Il permet de plus de prendre en charge la variabilité non résolue au fur et à mesure de l'exécution d'un processus.

T4VASP offre également des avantages dans la manière dont il met en œuvre l'approche proposée. Ainsi, au moment de la liaison des CA à la ligne de processus, T4VASP permet à l'architecte de spécifier l'étape (initialisation, exécution ou finalisation) d'une unité de travail qu'une liste de CA automatise, ainsi que l'ordre d'exécution de ces CA. Cela permet de définir plus finement l'automatisation des processus et également d'assurer que les initialisations et finalisations des unités de travail ne sont pas omises. T4VASP offre de plus un support à la résolution de la variabilité, i) en abstrayant le VAM des informations non pertinentes pour la personne en charge de la résolution de la variabilité, ii) en forçant le respect des contraintes entre les spécifications de variabilité et iii) en assurant le respect de la spécification de CVL. Au moment de l'exécution d'un processus, T4VASP offre un mécanisme permettant de lancer les CA qui automatisent ce processus de manière générique, ce qui en fait un outil indépendant des CA associés à une ligne de processus. De plus, les CA peuvent avoir accès aux informations contextuelles dont ils ont besoin pour s'exécuter, y compris celles qui ne peuvent pas être capturées par le modèle du processus en cours d'exécution. En effet, T4VASP demande ces informations contextuelles à l'acteur courant du processus et les sauvegarde dans un modèle contexte. Toutefois, afin que l'acteur courant du processus puisse se concentrer sur d'autres tâches pendant l'exécution d'une liste de CA, toutes les informations contextuelles nécessaires sont demandées au début de l'exécution de la liste. Pendant l'exécution d'un processus, T4VASP informe l'acteur

courant du processus lorsque des interventions manuelles sont requises de sa part, afin d'assurer le bon déroulement de l'exécution. Enfin, T4VASP permet à l'acteur courant du processus de suivre l'exécution du processus, car c'est cet acteur courant qui lance les initialisations, exécutions et finalisations des unités de travail.

Nous avons également identifié plusieurs extensions à T4VASP. Ainsi, faire évoluer T4VASP vers une application client-serveur permettrait l'exécution de processus en environnements distribués, ce qui correspond à la majorité des contextes de réalisation de projets de développement logiciel. D'autre part, spécifier, dans le modèle de CA abstraits et de liaisons, qu'une initialisation ou une finalisation d'unité de travail est manuelle permettrait que l'acteur courant d'un processus puisse les lancer même si elles ne sont pas automatisées. Enfin, en faisant évoluer le métamodèle de CA abstraits et de liaisons de manière à pouvoir spécifier que des listes d'AC sont entrecoupées de tâches manuelles, les utilisateurs d'un modèle de processus auraient la liberté de le définir au niveau d'abstraction qui leur convient le mieux. Ils pourraient en effet définir des unités de travail composées à la fois de tâches manuelles et automatisées.

Nous avons actuellement implémenté uniquement les fonctionnalités de T4VASP dont nous avions besoin pour pouvoir l'utiliser avec les cas d'application présentés dans le chapitre suivant. Afin d'assurer que T4VASP soit utilisable quel que soit le cas d'application, il reste donc à implémenter les fonctionnalités suivantes :

- prise en compte de l'ensemble de CVL par l'assistant à la résolution de la variabilité et par le moteur de dérivation,
- évaluation dynamique des expressions Kermeta,
- évaluation des conditions (liées aux AC ou aux unités de travail automatisées) définies dans un modèle de CA abstraits et de liaisons,
- découplage de l'accès aux informations contextuelles de l'implémentation des CA,
- résolution de la variabilité à l'exécution.

Nous évaluons le degré de maturité de T4VASP au niveau TRL (*Technology Readiness Level*) 2, où le TRL est une mesure permettant d'évaluer la maturité d'une technologie [oD11]. Au niveau TRL 2, les concepts et des applications d'une technologie ont été définis, mais la technologie n'est pas encore prête à être utilisée sur n'importe quel cas d'application.

Le code source de T4VASP peut être consulté à l'adresse suivante : `https://github.com/emmanuellerouille/T4VASP/tree/master/source_code`.

Chapitre 7

Applications de l'approche

À ce jour, T4VASP a été appliqué à deux cas : la famille de processus de métamodélisation qui sert d'exemple illustratif tout au long de cette thèse, ainsi qu'une famille simplifiée de processus de développement web Java issue de Sodifrance. Pour ce faire, nous avons pour chacun de ces deux cas défini une ligne de processus de développement logiciel en nous appuyant sur CVL4SP (section 4.2). Nous avons donc spécifié dans un VAM la variabilité au niveau des exigences des projets, défini un modèle de processus de base afin de capturer les éléments de modèle nécessaires à la création des différents processus de la famille, et défini un VRM afin de spécifier comment dériver un processus résolu du modèle de processus de base en fonction des exigences des projets. Nous avons utilisé l'outil SPEM-Designer pour modéliser les processus de base. Nous avons ensuite identifié et implémenté des CA pour chaque cas d'application en suivant M4RAC (section 5.2). Nous avons pour ce faire créé un modèle de CA abstraits et de liaisons pour chacuns des deux cas d'application. Toujours pour chacun des deux cas, un modèle de processus résolu a été dérivé en fonction d'une sélection d'exigences. Ce modèle de processus a ensuite été exécuté, et les CA qui lui sont liés ont été lancés au fur et à mesure de cette exécution. Nous détaillons dans la suite de ce chapitre les deux cas d'application. Les sections 7.1 et 7.2 présentent respectivement l'application à la famille de processus de métamodélisation et l'application à la famille de processus de développement web Java.

7.1 Application sur une famille de processus de métamodélisation

Nous appliquons dans cette section T4VASP à une famille de processus de métamodélisation, issue de l'équipe de recherche Triskell. Il s'agit de la famille qui a servi d'exemple illustratif à CVL4SP et T4VASP. Nous invitons donc le lecteur à se référer aux sections 4.1 et 6.1 pour une description complète de cette famille de processus et des tâches qui ont lieu durant l'exécution des processus de cette famille et qui gagneraient à être automatisées.

Le VAM de la famille de processus de métamodélisation contient 39 éléments de modèle. Ce VAM permet de capturer les exigences de 128 projets différents, chacun ayant un processus différent. Le modèle de processus de base ainsi que le VRM de la famille de processus de métamodélisation contiennent quant à eux respectivement 134 et 123 éléments de modèle. Le VAM, le modèle de processus de base, ainsi que le VRM forment donc une ligne de processus permettant de capturer 128 processus différents.

Le modèle de CA abstraits et de liaisons contient 27 éléments de modèle. Nous avons implémenté 10 CA en Java :

1. un CA permettant de créer un nouveau projet EMF avec un nouveau fichier Ecore vide,
2. un CA permettant de valider un métamodèle Ecore,
3. un CA permettant de générer un éditeur arborescent,
4. un CA permettant de créer un nouveau projet XText,
5. un CA permettant de créer un nouveau projet Kermeta et de l'initialiser avec un appel à une fonction Kermeta qui vérifie qu'un modèle respecte bien les contraintes définies sur son métamodèle,
6. un CA permettant de créer un nouveau projet Kermeta et de l'initialiser avec la génération du patron de conception interpréteur pour chaque méta-classe d'un métamodèle,
7. un CA permettant de créer un nouveau projet Kermeta et de l'initialiser avec la génération du patron de conception visiteur pour chaque méta-classe d'un métamodèle,
8. un CA permettant de connecter un espace de travail Eclipse à un dépôt SVN distant,
9. un CA permettant de mettre un projet sous contrôle de version,
10. un CA permettant de propager avec SVN du code source vers un dépôt distant.

Le CA 1 initialise la définition d'un métamodèle. Il a besoin de deux informations contextuelles pour s'exécuter : le nom du projet EMF et le nom du fichier Ecore. Le gestionnaire d'informations contextuelles les demande à l'acteur courant du processus puis les sauvegarde dans le modèle de contexte.

Le CA 2 finalise la définition d'un métamodèle et le CA 3 exécute la tâche de définition d'un éditeur arborescent. Ces deux CA ont besoin pour cela des deux mêmes informations contextuelles : le nom du projet EMF et du fichier Ecore contenant la définition du métamodèle. Le gestionnaire d'informations contextuelles trouve ces informations dans le modèle de contexte, où elles ont été préalablement écrites.

Le CA 4 initialise la définition d'un éditeur textuel, lorsque celle-ci est réalisée avec l'outil XText. Ce CA utilise l'information contextuelle correspondant au nom du projet EMF contenant la définition du métamodèle. Cette information est récupérée dans le modèle de contexte par le gestionnaire d'informations contextuelles. Ce CA nécessite également une autre information contextuelle pour s'exécuter, à savoir le

```java
public class GenerateTreeEditorActivityAutomation implements
        ActivityAutomation {

    public GenerateTreeEditorActivityAutomation() {
        // TODO Auto-generated constructor stub
    }

    @Override
    public void run(String contextModelPath) {
        GenerateTreeEditor createTreeEditor = new GenerateTreeEditor();
        createTreeEditor.run(
                PlatformUI.getWorkbench().getActiveWorkbenchWindow().getShell(),
                contextModelPath
        );
    }

}
```

FIGURE 7.1 – Point d'entrée du CA permettant de générer un éditeur arborescent

nom du projet XText à créer. Le gestionnaire d'informations contextuelles demande cette information à l'acteur courant du processus.

Les CA 5, 6 et 7 initialisent respectivement les tâches de définition d'un vérificateur, d'un interpréteur et d'un compilateur. Ils ont besoin tous les trois pour s'exécuter du nom du projet Kermeta à créer, ainsi que du nom du projet EMF et du fichier Ecore contenant la définition du métamodèle. Le gestionnaire d'informations contextuelles demande le nom du projet à créer à l'acteur courant du processus et récupère dans le modèle de contexte le nom du projet EMF et du fichier Ecore contenant la définition du métamodèle. La création d'un projet Kermeta est commune à ces trois CA et n'est pas factorisée dans son propre CA. En effet, les initialisations des projets Kermeta (appel à une fonction Kermeta vérifiant qu'un modèle respecte bien les contraintes définies sur son métamodèle et génération du patron de conception interpréteur ou compilateur) ont besoin pour s'exécuter de connaître le nom du projet à initialiser, qui est différent dans les trois cas. Il faut donc pouvoir différencier les noms des projets Kermeta. De plus, le nom d'un projet Kermeta ne peut pas être écrasé dans le modèle de contexte car il est réutilisé dans le cas de l'interpréteur pour sa mise sous contrôle de version (et l'information pourrait être écrasée avant que la mise sous contrôle de version n'ait lieu car les tâches de définition d'un interpréteur, d'un compilateur et d'un vérificateur s'exécutent en parallèle). Nous avons vu, dans la section 6.3.3, que dans ce cas des CA différents automatisent la même TMR, mais en demandant chacun des informations contextuelles spécifiques aux artéfacts qu'ils impactent.

Les CA 8, 9 et 10 finalisent la tâche de définition d'un interpréteur. Le CA 8 a besoin pour s'exécuter de l'URL d'un dépôt SVN distant, ainsi que d'un nom d'utilisateur et d'un mot de passe pour se connecter à ce dépôt distant. Le gestionnaire d'informations contextuelles demande ces trois informations à l'acteur courant du processus. Les CA 9 et 10 ont besoin pour s'exécuter du nom du projet dont le contrôle de version doit être géré (ici le projet Kermeta contenant le code source de l'interpréteur). Le CA 9 a de plus besoin de connaître l'URL du dépôt SVN distant. Le gestionnaire d'informations contextuelles récupère toutes ces informations dans le modèle de contexte. Ici,

```
51  public void run(Shell shell, String contextModelPath){
52
53      String projectName = null;
54      String ecoreModelName = null;
55
56      //Read the project name and the ecore model name
57      // Initialize the model
58      ExecutioncontextPackage.eINSTANCE.eClass();
59
60      // Register the XMI resource factory for the .executioncontext extension
61
62      Resource.Factory.Registry reg = Resource.Factory.Registry.INSTANCE;
63      Map<String, Object> m = reg.getExtensionToFactoryMap();
64      m.put("executioncontext", new XMIResourceFactoryImpl());
65
66      // Obtain a new resource set
67      ResourceSet resSet = new ResourceSetImpl();
68
69      // Get the resource
70      Resource resource = resSet.getResource(URI
71              .createURI(contextModelPath), true);
72      // Get the first model element and cast it to the right type, in my
73      // example everything is hierarchical included in this first node
74      ExecutionContext executionContext = (ExecutionContext) resource.getContents().get(0);
75
76      for(Key key: executionContext.getKeys()){
77          if(key.getName().equals("project name")){
78              projectName = key.getValue().getContent();
79          }
80          if(key.getName().equals("model file name")){
81              ecoreModelName = key.getValue().getContent();
82          }
83      }
84
85      //get object which represents the workspace
86      IWorkspace workspace = ResourcesPlugin.getWorkspace();
87
88      //get location of workspace (java.io.File)
89      File workspaceDirectory = workspace.getRoot().getLocation().toFile();
90
91      GenModelHelper genModelHelper = new GenModelHelper(projectName);
92      genModelHelper.createGenModel(java.net.URI.create("/"+projectName+"/model/"+ecoreModelName+".ecore"),
93              new File(workspaceDirectory+"/"+projectName+"//model//"+ecoreModelName+".genmodel"),
94              new File(workspaceDirectory+"/"+projectName+"//src"), new Boolean(false));
95  }
```

FIGURE 7.2 – Méthode `run` de la classe `GenerateTreeEditor`, appelée par la méthode `run` de la figure 7.1

l'automatisation de la propagation avec SVN du code source vers un dépôt distant est mise dans son propre CA (ici le 10), afin de pouvoir la réutiliser indépendamment des CA 8 et 9. En effet, la première fois que la tâche de définition d'un interpréteur a lieu, il faut, lors de sa finalisation, exécuter les CA 8, 9 et 10, alors que les fois suivantes, seul le CA 10 doit être exécuté. Nous avons implémenté l'automatisation de la connexion d'un workspace Eclipse à un dépôt SVN distant et la mise d'un projet sous contrôle de version, dans des CA séparés, les CA 8 et 9, afin de séparer les préoccupations.

Les figures 7.1 à 7.4 présentent des extraits de code des CA. C'est la méthode `run` de la figure 7.1 qui est appelée pour exécuter le CA 3, qui permet de générer un éditeur arborescent. Cette méthode fait elle-même appel à la méthode `run` de la figure 7.1, qui contient le code générant l'éditeur arborescent. Dans un premier temps (lignes 53 à 83) les noms du projet EMF et du fichier Ecore contenant la définition du métamodèle sont récupérés dans le modèle de contexte. Rappelons à cette occasion que comme

```
public class CreateKermetaInterpreterProjectActivityAutomation implements
        ActivityAutomation {

    public CreateKermetaInterpreterProjectActivityAutomation() {
        // TODO Auto-generated constructor stub
    }

    @Override
    public void run(String contextModelPath) {
        CreateKermetaInterpreterProject createKermetaInterpreterProject =
                new CreateKermetaInterpreterProject();
        createKermetaInterpreterProject.run(
                PlatformUI.getWorkbench().getActiveWorkbenchWindow(),
                contextModelPath);
    }

}
```

FIGURE 7.3 – Point d'entrée du CA initialisant la tâche de définition d'un interpréteur

indiqué en section 6.5, le découplage de l'accès aux informations contextuelles de l'implémentation du CA, grâce au gestionnaire d'informations contextuelles, n'a pas encore été implémenté. Le code restant (de la ligne 83 à la fin) permet quant à lui de générer l'éditeur arborescent, en fonction du nom du projet EMF et du fichier Ecore précédemment récupérés. La méthode `run` de la figure 7.3 est appelée pour exécuter le CA 6, qui crée un nouveau projet Kermeta et l'initialise avec la génération du patron de conception interpréteur. Cette méthode appelle la méthode `run` de la figure 7.4. Cette dernière méthode commence par créer un nouveau projet Kermeta (lignes 59 à 67), puis elle appelle la méthode `generateInterpreter` (ligne 69). Cette dernière génère dans le projet Kermeta nouvellement créé le patron interpréteur pour chaque métaclasse d'un métamodèle, dont le nom du projet EMF et du fichier Ecore correspondant sont récupérés dans le modèle de contexte (cf. lignes 75 à 79). Le début de la méthode `generateInterpreter` est présenté à partir de la ligne 72.

Pour résoudre la variabilité, au maximum 7 choix sont à faire dans ce cas. Il faut en effet déterminer si un interpréteur, un compilateur, un éditeur textuel, un éditeur arborescent ou un vérificateur sont requis pour un projet donné. Dans le cas ou un interpréteur (respectivement un éditeur textuel) est requis, la personne en charge de la résolution de la variabilité doit également déterminer si c'est l'outil SVN ou Git (respectivement XText ou EMFText) doit être utilisé. Dans le cadre de ce cas d'application, nous avons sélectionné un interpréteur, un compilateur, un éditeur arborescent ainsi qu'un vérificateur. Nous avons également sélectionné l'outil SVN comme SCV. La figure 7.5 illustre le processus résolu correspondant.

```
57    public void run(IWorkbenchWindow window, String contextModelPath){
58        //Create a new Kermeta project
59        try{
60            KermetaProjectNewWizard kermetaProjectNewWizard =
61                    new KermetaProjectNewWizard(contextModelPath);
62            kermetaProjectNewWizard.init(window.getWorkbench(), selection);
63            WizardDialog dialog = new WizardDialog(window.getShell(), kermetaProjectNewWizard);
64            dialog.open();
65        }catch(Exception e){
66            e.printStackTrace();
67        }
68
69        generateInterpreter(contextModelPath);
70    }
71
72    public void generateInterpreter(String contextModelPath)
73    {
74        //Get the root of the context model
75        ExecutionContext executionContext = ModelUtils.getExecutionContextRoot(contextModelPath);
76
77        //Initialize the needed variables
78        String projectName = ModelUtils.getValueOfKey("project name", executionContext);
79        String metamodelName = ModelUtils.getValueOfKey("model file name", executionContext);
80        String interpreterProjectName =
81                ModelUtils.getValueOfKey("interpreter name", executionContext);
82
83        //Load the root of the metamodel
84        ResourceSet resourceSet = new ResourceSetImpl();
85        Resource ecoreResource = resourceSet.getResource(
86                URI.createURI("platform:/resource/"+projectName+"/model/"+metamodelName+".ecore"),
87                true);
88        EPackage root = (EPackage)ecoreResource.getContents().get(0);
89
90        //Create files in which the code will be generated
91        System.out.println(ResourcesPlugin.getWorkspace().getRoot().getLocation().toFile());
92        File visitorFile = new File(
93                ResourcesPlugin.getWorkspace().getRoot().getLocation().toFile()+"/"
94                        +interpreterProjectName+"/src/main/kmt/"+metamodelName+"_context.kmt");
95        File elementsFile = new File(
96                ResourcesPlugin.getWorkspace().getRoot().getLocation().toFile()+"/"
97                        +interpreterProjectName+"/src/main/kmt/"+metamodelName+"_eval.kmt");
98
99        if (!visitorFile.exists()) {
100           try {
101               visitorFile.createNewFile();
```

FIGURE 7.4 – Extrait des méthodes permettant l'initialisation d'un interpréteur Kermeta

Une démonstration du cas d'application de la famille de processus de métamodélisation peut être visualisée à l'adresse suivante : http://youtu.be/NYZT5cA5-cY. Les modèles ainsi que les CA relatifs à ce cas d'utilisation peuvent être consultés sur le dépôt SVN suivant : http://t4vasp.googlecode.com/svn/trunk/application_metamodeling_process_line. Nous en présentons un extrait dans l'annexe A.

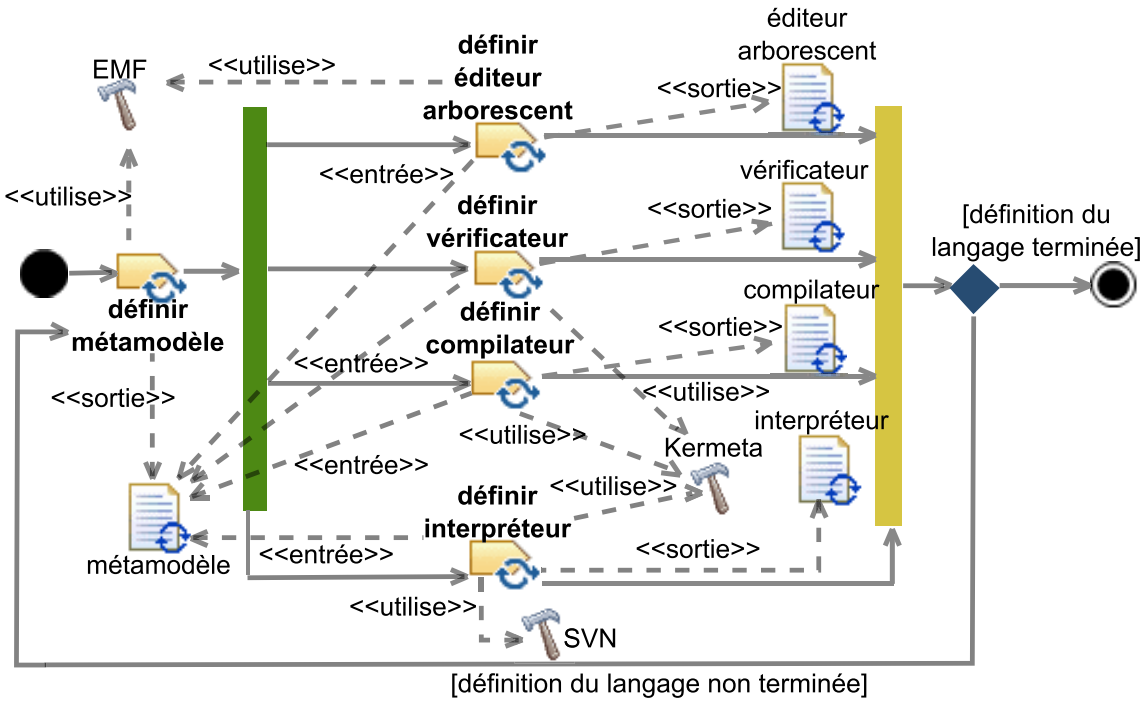
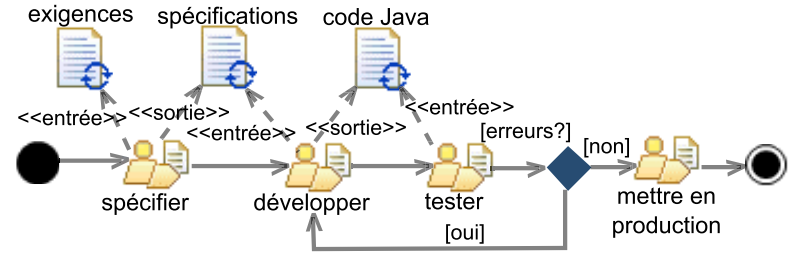


FIGURE 7.5 – Modèle de processus résolu du cas d'application de la famille de processus de métamodélisation

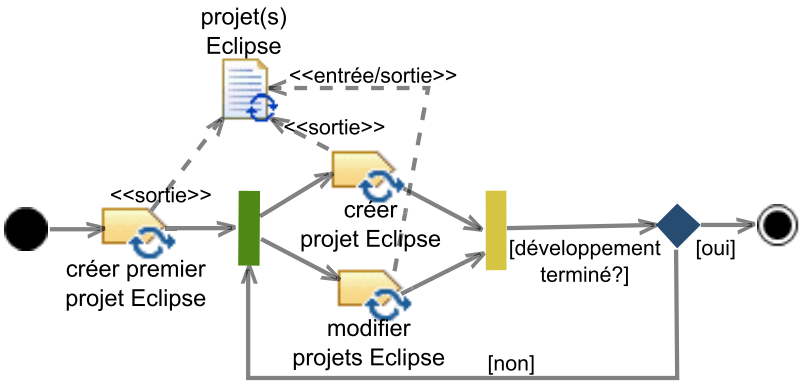
7.2 Application sur une famille de processus de développement web Java

Nous présentons dans cette section l'application de T4VASP à une famille simplifiée de processus de développement d'une application web en Java, issue de Sodifrance.

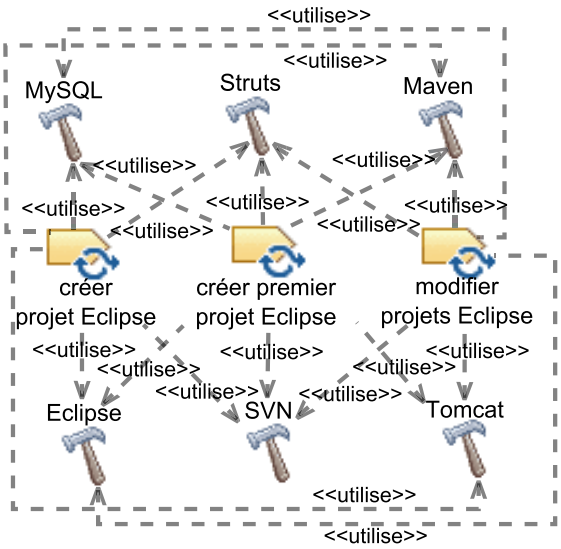
Nous commençons par présenter cette famille. Pour ce faire, nous introduisons d'abord un exemple simplifié de processus de développement web Java, illustré dans la figure 7.6. Nous détaillons ensuite des exemples de variantes de ce processus ainsi que les TMR qui ont lieu durant l'exécution des processus de cette famille et qui gagneraient à être automatisées. La sous-figure 7.6(a) montre le flot d'activités du processus de développement web Java illustré dans la figure 7.6, ainsi que les produits de travail consommés et produits par ces activités. L'exemple de processus de développement web Java présenté commence avec l'activité *spécifier*. Durant cette activité, un client produit un document définissant les spécifications techniques et fonctionnelles de l'application à développer, en fonction des exigences d'un projet. Vient ensuite l'activité *développer*, qui consiste à coder manuellement l'application à développer. L'activité *tester* suit, durant laquelle des développeurs testent l'application en cours de développement. Si cette activité révèle des erreurs dans le code de l'application, celles-ci doivent être corrigées, ce qui se traduit dans le processus par un retour à l'activité de développement. S'il n'y a pas d'erreurs, le processus se termine par l'activité *mettre en*



(a) Flot d'activités et produits de travail associés



(b) Détail de l'activité *développer*



(c) Outils utilisés pendant l'activité *développer*

FIGURE 7.6 – Un exemple simplifié de processus de développement web Java

production, qui consiste à déployer l'application sur l'environnement du client.

La sous-figure 7.6(b) détaille l'activité de développement. Celle-ci consiste, dans le cas de cet exemple, en la création et la modification de projets Eclipse. La sous-

figure 7.6(c) détaille les outils utilisés lors de l'activité de développement. L'environnement de développement est Eclipse, le SCV est SVN, le serveur web est Tomcat [1], la base de données est MySQL [2], le *framework* pour l'IHM web est Struts [3] et l'outil de compilation est Maven [4].

Il existe des variantes de ce processus, dont quelques exemples sont présentés dans ce paragraphe. Ainsi, si le client fournit les spécifications dans un langage interprétable par un ordinateur (par exemple UML 2), alors une activité de génération de code a lieu avant l'activité de développement. L'activité de génération de code consiste à générer le code Java qui est similaire pour toutes les entités de l'application à développer. Lorsque l'activité de génération de code a lieu, alors l'activité de développement consiste juste à rajouter manuellement le code qui n'a pas pu être généré. L'activité de mise en production peut également prendre des formes différentes. En effet, pendant cette activité, selon les exigences du client, la livraison de l'application peut se faire en livrant uniquement le code source de l'application, en livrant uniquement le code compilé de l'application, en livrant les deux (code source et code compilé), ou un ingénieur de Sodifrance peut aller chez le client pour installer lui-même l'application. Les exigences des projets peuvent également motiver l'utilisation d'outils autres que ceux présentés dans la sous-figure 7.6(c). Par exemple, une base de données Oracle [5] ou Postgresql [6] peut être utilisée à la place de la base de données MySQL, et il peut même ne pas y avoir de base de données s'il n'y a pas de donnée à persister. De même, Ant [7] peut être utilisé à la place de Maven, et un *framework* autre que Struts peut être utilisé pour l'IHM web, tel que JSF [8] (Java Server Faces), Flex [9] ou GWT [10] (Google Web Toolkit).

Des TMR qui gagneraient à être automatisées ont lieu durant l'exécution des processus de cette famille. Nous en présentons ici quelques exemples. Ainsi, lors de l'initialisation de l'activité *développer*, chaque développeur et architecte doit configurer son environnement de travail Eclipse. Cette configuration consiste, entre autres, à installer les plug-ins Eclipse WTP [11] et Subclipse [12] (plug-ins permettant respectivement d'utiliser Tomcat et SVN dans Eclipse), à créer dans l'environnement de travail Eclipse un nouveau serveur Tomcat et à connecter l'espace de travail Eclipse à un dépôt SVN distant. Lors de l'initialisation de l'activité de développement, un architecte est de plus en charge de créer un nouveau dossier sur ce dépôt distant, qui contiendra le code de l'application à développer. La création d'un projet Eclipse se termine toujours par sa

1. http://tomcat.apache.org/
2. http://www.mysql.com/
3. http://struts.apache.org/
4. http://maven.apache.org/
5. http://www.oracle.com/us/products/database/overview/index.html
6. http://www.postgresql.org/
7. http://ant.apache.org/
8. http://www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html
9. http://www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html
10. http://www.gwtproject.org/
11. http://www.eclipse.org/webtools/
12. http://subclipse.tigris.org/

mise sous contrôle de version, puis par sa propagation vers le dépôt SVN distant. En particulier, seules les sources contenues par ce projet doivent être mises sous contrôle de version, et pas les fichiers binaires contenant le code Java compilé, afin d'éviter de stocker des données inutiles sur le dépôt distant. La modification d'un projet Eclipse se termine toujours par la propagation des modifications sur le dépôt SVN distant.

Le VAM de la famille de processus de développement web Java contient 27 éléments de modèle. Ce VAM permet de capturer les exigences de 512 projets différents, chacun ayant un processus différent. Le modèle de processus de base ainsi que le VRM de la famille de processus de développement web Java contiennent quant à eux respectivement 100 et 44 éléments de modèle. Le VAM, le modèle de processus de base ainsi que le VRM forment donc une ligne de processus capturant 512 processus différents.

Le modèle de CA abstraits et de liaisons contient 25 éléments de modèle. Nous avons implémenté 9 CA en Java :

1. un CA permettant d'installer le plug-in Eclipse WTP,
2. un CA permettant d'installer le plug-in Eclipse Subclipse,
3. un CA permettant de redémarrer Eclipse,
4. un CA permettant de créer un nouveau serveur Tomcat,
5. un CA permettant de connecter un espace de travail Eclipse à un dépôt SVN distant,
6. un CA permettant de créer un nouveau dossier sur un dépôt SVN distant,
7. un CA permettant de mettre un projet Eclipse sous contrôle de version avec SVN (ce CA ne met sous contrôle de version que les sources, pas les fichiers binaires),
8. un CA permettant de propager avec SVN le code source d'un projet Eclipse spécifique vers un dépôt distant,
9. un CA permettant de propager vers un dépôt SVN distant particulier le code source de tous les projets Eclipse partagés sur ce dépôt, lorsque ceux-ci ont subit des modifications.

Les CA 1 à 6 initialisent l'activité de développement. Le CA 1 a besoin pour s'exécuter du chemin, sur le poste local d'un développeur ou d'un architecte, de l'environnement Eclipse sur lequel installer le plug-in WTP. Le gestionnaire d'informations contextuelles demande cette information à l'acteur courant du processus et la sauvegarde dans le modèle de contexte. Le CA 2 a besoin pour s'exécuter de la même information contextuelle, que le gestionnaire d'informations contextuelles trouve donc dans le modèle de contexte. L'intérêt du CA 3 est de déployer dans l'environnement Eclipse les plug-ins nouvellement installés. Ce CA, tout comme le CA 4, n'a pas besoin d'informations contextuelles pour s'exécuter. Le CA 5 a besoin de trois informations contextuelles pour s'exécuter : l'URL d'un dépôt SVN distant, ainsi qu'un nom d'utilisateur et un mot de passe pour se connecter à ce dépôt distant. Le gestionnaire d'informations contextuelles demande ces trois informations à l'acteur courant du processus. Afin d'exécuter le CA 6, le gestionnaire d'informations contextuelles demande à un architecte le nom du dossier à créer sur le dépôt distant et récupère dans

le modèle de contexte l'URL de ce dépôt distant. Afin de séparer les préoccupations, nous avons implémenté dans des CA différents (à savoir les CA 1 à 6) l'automatisation de l'installation des plug-ins WTP et Subclipse, le redémarrage d'Eclipse, la création d'un nouveau serveur Tomcat, la connection d'un espace de travail Eclipse à un dépôt SVN distant et la création d'un nouveau dossier sur un dépôt SVN distant.

Les CA 7 et 8 finalisent la tâche de création d'un nouveau projet Eclipse. Le CA 7 a besoin de deux informations contextuelles pour s'exécuter : le nom du projet Eclipse à mettre sous contrôle de version et l'URL du dépôt distant. Le gestionnaire d'informations contextuelles demande la première information à l'acteur courant du processus, à chaque fois que le CA 7 est appelé, et sauve l'information dans le modèle de contexte en écrasant son ancienne valeur si elle existe. Cette information est en effet différente pour chaque projet, les projets à créer ne sont pas forcément connus à l'avance (il n'est donc pas possible de créer un CA spécifique à chaque projet) et l'information n'est pas réutilisée après l'exécution des CA 7 et 8 (le fait qu'elle soit écrasée ne pose donc pas de problème). Toujours pour le CA 7, le gestionnaire d'informations contextuelles récupère l'URL du dépôt distant dans le modèle de contexte. Pour le CA 8, le gestionnaire d'informations contextuelles récupère dans le modèle de contexte le nom du projet Eclipse à mettre sous contrôle de version. Nous avons implémenté dans deux CA différents (les CA 7 et 8) l'automatisation de la mise sous contrôle de version d'un projet Eclipse et la propagation de ce projet vers un dépôt SVN distant afin de séparer les préoccupations.

```
 9 public class InstallSubclipse implements ActivityAutomation {
10
11     public InstallSubclipse() {
12         // TODO Auto-generated constructor stub
13     }
14
15     @Override
16     public void run(String contextModelPath) {
17         ExecutionContext executionContext =
18                 ModelUtils.getExecutionContextRoot(contextModelPath);
19         String localPathOfTheCurrentEclipseApplication =
20                 ModelUtils.getValueOfKey(
21                         "local path of the current Eclipse application",
22                         executionContext);
23         String localPathOfTheCurrentEclipseApplicationAfterConversion =
24                 StringUtils.convertPathIntoString(
25                         localPathOfTheCurrentEclipseApplication);
26         String cmd =
27                 localPathOfTheCurrentEclipseApplicationAfterConversion+
28                 " -application org.eclipse.equinox.p2.director "
29                 +"-repository http://subclipse.tigris.org/update_1.8.x "
30                 +"-installIU org.tigris.subversion.subclipse.feature.group/1.8.22,"
31                 +"org.tigris.subversion.clientadapter.feature.feature.group/1.8.6,"
32                 +"org.tigris.subversion.clientadapter.javahl.feature.feature.group/1.7.10";
33         RunCommand.runCommand(cmd);
34     }
35
36 }
```

FIGURE 7.7 – Code du CA permettant d'installer le plug-in Eclipse Subclipse

Enfin, le CA 9 finalise la tâche de modification de projets Eclipse. Il a besoin pour

```java
public class NewTomcatServer implements ActivityAutomation {

    public NewTomcatServer() {
        // TODO Auto-generated constructor stub
    }

    @Override
    //Creation of a new and preconfigured Tomcat server
    public void run(String contextModelPath) {

        //Creation of a Tomcat runtime configuration
        IRuntimeType[] runtimeTypes = ServerCore.getRuntimeTypes();
        for(IRuntimeType runtimeType : runtimeTypes){
            if(runtimeType.getName().equals("Apache Tomcat v7.0")){
                try {
                    IRuntimeWorkingCopy runtimeWorkingCopy =
                            runtimeType.createRuntime(null, new NullProgressMonitor());
                    runtimeWorkingCopy.setLocation(
                            new Path("C:\\Program Files\\Apache Software Foundation\\Tomcat 7.0"));
                    IStatus status = runtimeWorkingCopy.validate(new NullProgressMonitor());
                    System.out.println("status code = "+status.getCode());
                    System.out.println("status message = "+status.getMessage());
                    runtimeWorkingCopy.save(false, new NullProgressMonitor());
                    System.out.println("save done");
                } catch (CoreException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
        }

        //Save the newly created runtime
        IRuntime myRuntime = null;
        IRuntime[] runtimes = ServerCore.getRuntimes();
        for(IRuntime runtime : runtimes){
            if(runtime.getName().equals("Apache Tomcat v7.0")){
                myRuntime = runtime;
            }
        }

        //Creation of a new server with the newly created runtime configuration
        IServerType[] serverTypes = ServerCore.getServerTypes();
        for(IServerType serverType : serverTypes){
            if(serverType.getName().equals("Tomcat v7.0 Server")){
                try {
                    IServerWorkingCopy serverWorkingCopy =
                            serverType.createServer(null, null, myRuntime, null);
                    serverWorkingCopy.save(false, new NullProgressMonitor());
                } catch (CoreException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
        }

    }

```

FIGURE 7.8 – Code du CA créant un nouveau serveur Tomcat

s'exécuter de l'URL du dépôt SVN distant. Le gestionnaire d'informations contextuelles récupère cette information dans le modèle de contexte.

Les figures 7.7 et 7.8 présentent l'implémentation des CA 2 et 4 respectivement.

L'implémentation du CA 2 consiste à exécuter une commande Windows installant, sur un environnement Eclipse,les archives Java relatives au plug-in Subclipse. L'implémentation du CA 4 consiste quant à elle à créer et sauvegarder un serveur Tomcat et une configuration associée.

Pour résoudre la variabilité, la personne en charge de cette résolution doit faire au maximum 7 choix : déterminer le SCV, si une base de données est utilisée, et si oui de quel type de base de données il s'agit, le *framework* pour l'IHM, l'outil de compilation, si une partie du code est générée et le mode de livraison. Pour ce cas d'application nous avons sélectionné un processus avec SVN comme SCV, MySQL comme base de données, Struts comme *framework* pour l'IHM, Maven comme outil de compilation, sans génération de code, et avec livraison à la fois du code source et du code compilé. Le processus résolu correspondant est donc le même que celui de la figure 7.6.

> Une démonstration du cas d'application de la famille de processus de développement web Java peut être visualisée à l'adresse suivante : http://youtu.be/71shRD6ax9k. Les modèles ainsi que les CA relatifs à ce cas d'utilisation peuvent être consultés sur le dépôt SVN suivant : http://t4vasp.googlecode.com/svn/trunk/application_java_development_process_line. Nous en présentons un extrait dans l'annexe B.

7.3 Synthèse et discussion

Ces applications montrent que notre approche permet de gérer la variabilité et d'automatiser des exemples réels de processus, qui ont de nombreuses variantes ainsi que de nombreuses tâches récurrentes qui gagnent à être automatisées. Le tableau 7.1 récapitule le nombre d'éléments de modèle et de CA nécessaires à la réalisation de chacun des deux cas d'application.

Cas d'application	nb élts processus de base	nb élts VAM	nb élts VRM	nb élts modèle CA abstraits et liaisons	nb CA	nb processus dans la famille
Famille processus métamodélisation	134	39	123	27	10	128
Famille processus développement web Java	100	27	44	25	9	512

TABLE 7.1 – Tableau récapitulatif du nombre d'éléments de modèle et de CA pour chaque cas d'application

Ces applications ont également permis d'observer des points forts de notre approche. Premièrement, pour les deux cas d'application, la modélisation des familles

de processus avec CVL4SP s'est révélée être plus économique, en termes de nombre d'éléments à modéliser, que la modélisation en extension de ces familles. En effet, pour la famille de processus de métamodélisation, la ligne de processus est constituée de 296 éléments de modèle et capture 128 processus différents. La modélisation en extension de ces 128 processus aurait nécessité au moins 5248 éléments de modèle. En effet, la variante de processus de métamodélisation qui contient le moins d'éléments de modèle est celle qui consiste uniquement en la définition d'un métamodèle, et dans ce cas 41 éléments de processus sont nécessaires pour modéliser cette variante. Dans le cas de la famille de processus de développement web Java, la ligne de processus est constituée de 171 éléments de modèle et elle capture 512 processus différents. La modélisation en extension de ces 512 processus aurait nécessité au moins 41472 éléments de modèle. En effet, les variantes de processus qui contiennent le moins d'éléments de modèle sont celles sans génération de code et sans base de données, indépendamment des outils utilisés pour le SCV, l'IHM et la compilation, et indépendamment du mode de livraison considéré. Dans ce cas, 81 éléments de processus sont nécessaires pour modéliser une telle variante. Si, dans les deux cas d'application, la modélisation avec CVL4SP des familles de processus s'est révélée être plus économique que leur modélisation en extension, c'est parce que les familles sont modélisées en intention, et qu'assez d'éléments sont communs aux processus de chaque famille pour que leur factorisation permette de faire des économies de modélisation, même si cette factorisation implique de devoir modéliser plus d'éléments pour spécifier comment dériver un processus de la famille. D'une manière générale, quel que soit le cas d'application, plus les processus d'une famille ont d'éléments en commun, plus leur modélisation en intention permettra de réduire le nombre d'éléments à modéliser par rapport à la modélisation en extension. Le deuxième point fort observé est que l'automatisation des TMR permet de réduire l'occurrence des erreurs humaines. Par exemple, lors de la mise sous contrôle de version d'un projet Java Eclipse avec Subclipse, les fichiers binaires sont également ajoutés au contrôle de version par défaut. Il arrive que les personnes en charge de la mise sous contrôle de version d'un projet Java Eclipse oublient de retirer ces fichiers binaires du contrôle de version, ce qui n'arrive plus une fois que cette tâche est automatisée. Un troisième point fort observé lors de ces applications est que l'automatisation des TMR permet de gagner du temps lorsqu'une personne en charge de la réalisation d'une tâche ne sait pas comment la réaliser. En effet, l'automatisation d'une TMR permet à cette personne d'éviter de passer du temps à se former à la réalisation de cette tâche.

Ces applications ont aussi permis de mettre en évidence une faiblesse de l'approche proposée dans cette thèse. Ainsi, M4RAC ne s'applique qu'au niveau d'une famille de processus, et ne permet donc pas d'améliorer la réutilisation de CA entre familles. Par exemple, les CA permettant de connecter un espace de travail Eclipse à un dépôt SVN distant, de mettre un projet sous contrôle de version et de propager du code source vers un dépôt distant sont communs aux deux cas d'application présentés dans ce chapitre. Or, comme M4RAC ne s'applique que sur un seul cas d'application à la fois, elle ne permet pas d'identifier le niveau de réutilisation de ces CA afin qu'ils soient réutilisables pour les deux cas d'application. Élargir M4RAC à l'ensemble des familles

de processus d'une entreprise permettrait de surmonter cette limitation.

8

Conclusion et perspectives

Dans ce chapitre, nous faisons le bilan des contributions cette thèse (section 8.1) et nous discutons les perspectives ouvertes par ces contributions (section 8.2).

8.1 Conclusion

La complexité des logiciels et des projets de développement logiciel amène une multiplication des outils de développement logiciel afin de faire face à cette complexité. Par exemple, les systèmes de contrôle de version permettent de gérer les difficultés liées au travail collaboratif et géographiquement distribué, les outils de compilation, de test ou d'intégration continue supportent les méthodes de développement agiles, les IDE permettent d'intégrer les différentes technologies utilisées sur un projet, etc. De plus, des versions différentes de ces outils existent afin de satisfaire les besoins spécifiques à chaque projet, ce qui augmente le nombre d'outils existant. L'utilisation de tous ces outils est cependant à l'origine de nombreuses TMR (Tâches Manuelles Récurrentes), sources d'erreurs et coûteuses en temps.

L'automatisation des TMR permet de réduire les erreurs liées à leur réalisation et donc le temps passé à corriger ces erreurs. La réutilisation des automatisations de TMR permet quant à elle de réduire le temps passé à les définir. Mais cette réutilisation est difficile. En effet, une automatisation de TMR n'étant pas forcément utile pour tous les projets et pouvant avoir des dépendances avec d'autres tâches, pour quels projets et à quels moments d'un projet réutiliser une automatisation de TMR ? De plus, une automatisation de TMR pouvant être utile dans des projets différents, ou à des moments différents d'un même projet, comment s'assurer qu'elle soit bien réutilisable à travers ses différents cas d'utilisation ?

Dans la littérature, il n'y a, à notre connaissance, pas d'approche permettant à la fois de savoir pour quels projets réutiliser des automatisations de TMR et à quels moments d'un projet les réutiliser. De plus, même s'il existe des approches centrées sur la conception et le développement de composants logiciels réutilisables, celles-ci ne sont pas adaptées au cas des automatisations de TMR. En effet, elles sont spécifiques aux composants logiciel réalisant des produits logiciels et supposent que chaque caracté-

ristique d'un produit logiciel est réalisée par un composant. Or, toutes les étapes d'un processus ne sont pas forcément réalisées par des automatisations de TMR.

Afin de surmonter ces limitations, nous proposons dans cette thèse une approche pilotant l'automatisation des TMR par les processus de développement logiciel. Elle s'appuie sur une première sous-contribution, CVL4SP, afin de définir en intention une ligne de processus et d'en dériver un processus en fonction des exigences d'un projet. Des composants automatisant des TMR (CA) sont liés aux unités de travail de la ligne de processus qu'ils automatisent. L'approche proposée s'appuie sur une deuxième sous-contribution, M4RAC, afin d'améliorer la réutilisation de ces CA à travers les différentes unités de travail qu'ils automatisent. Un processus dérivé de la ligne de processus est automatisé en lançant, au fur et à mesure de son exécution, les CA qui lui sont liés. La réutilisation des CA passe donc par la réutilisation des processus de développement logiciel, le lien entre eux permettant de savoir pour quels projets et à quels moments d'un projet réutiliser ces CA. D'autre part, certaines contraintes de projet impliquant de commencer l'exécution d'un processus alors que sa variabilité n'est que partiellement résolue, l'approche proposée permet de résoudre de la variabilité lors de l'exécution d'un processus. Cela permet d'assurer le bon déroulement de l'exécution d'un processus même lorsque sa variabilité n'est que partiellement résolue.

CVL4SP s'appuie sur CVL afin de gérer la variabilité des processus de développement logiciel. Le point fort de CVL4SP, en comparaison des autres approches gérant la variabilité dans les processus, est son indépendance vis-à-vis du formalisme utilisé pour définir les processus.

M4RAC améliore l'identification du niveau de réutilisation des CA, c'est-à-dire des parties de ces CA qui varient et qui ne varient pas. Cette information est nécessaire à la création de CA réutilisables à travers leurs différents cas d'utilisation. M4RAC s'appuie sur le lien entre la ligne de processus et les CA afin d'identifier les différents contextes d'utilisation de ces derniers. Ce sont ces différents contextes d'utilisation qui guident l'identification du niveau de réutilisation d'un CA. L'avantage principal de M4RAC est qu'elle explicite les différents contextes d'utilisation de chaque CA, ce qui permet de ne pas en oublier et de ne pas prendre en compte des contextes d'utilisation inutiles.

Afin de démontrer la faisabilité de notre approche, nous avons développé un outil la supportant : T4VASP. Il a été appliqué à deux cas d'utilisation, à savoir une famille de processus de métamodélisation et une famille de processus de développement web Java issue de Sodifrance. T4VASP permet de définir en intention une ligne de processus, de lui lier des CA, d'en dériver un processus en fonction des exigences d'un projet, d'automatiser l'exécution de ce processus et de résoudre sa variabilité non résolue au moment de son exécution.

8.2 Perspectives

Les contributions de cette cette thèse ouvrent plusieurs perspectives de travail. Certaines sont d'ordre industriel, tandis que d'autres sont des pistes de recherche à

investiguer à court et long terme.

8.2.1 Perspectives industrielles

Nous présentons ici différents axes de travail afin de permettre l'utilisation de T4VASP dans un contexte industriel.

8.2.1.1 Utilisation de T4VASP quel que soit le cas d'application

T4VASP ne fonctionne actuellement que dans un environnement local. Or, les projets de développement logiciel requièrent souvent l'intervention de plusieurs personnes, chacune ayant son propre poste de travail. Aussi, faire évoluer T4VASP vers une application client-serveur permettrait de l'utiliser en environnements distribués.

Nous n'avons implémenté que les fonctionnalités de T4VASP permettant de l'utiliser avec les deux cas d'applications présentés dans cette thèse. L'implémentation des fonctionnalités restantes de T4VASP permettrait donc de pouvoir l'utiliser quel que soit le cas d'application. Les fonctionnalités non encore implémentées portent sur la prise en compte exhaustive de CVL par l'assistant à la résolution de la variabilité et par le moteur de dérivation, l'évaluation dynamique des expressions Kermeta, l'évaluation des conditions définies dans un modèle de CA abstraits et de liaisons, le découplage de l'accès aux informations contextuelles de l'implémentation des CA et la résolution de la variabilité à l'exécution.

Nous avons vu dans la section 4.3.4 que la modélisation des éléments de processus externes pouvait se révéler difficile lorsqu'ils ne respectent pas toutes les contraintes définies sur leur métamodèle et que le modeleur utilisé ne permet pas de modéliser des éléments invalides. L'ajout à T4VASP d'un composant permettant de générer automatiquement ces éléments, ainsi que les éléments de modèle manquants pour les rendre valides, permettrait de faire face à cette difficulté.

8.2.1.2 Limitation des erreurs

T4VASP ne permet actuellement pas à l'acteur courant du processus de lancer des initialisations ou des finalisations d'unités de travail lorsqu'elles ne sont pas automatisées. Elles sont alors traitées par l'interpréteur de processus comme si elles n'existaient pas. L'acteur courant du processus peut donc omettre de les réaliser. La solution idéale serait de pouvoir spécifier dans le modèle de processus qu'une unité de travail a une initialisation ou une finalisation, qu'elle soit manuelle ou non. Cela permettrait à l'interpréteur de processus de différencier les cas où une initialisation ou finalisation est manuelle de ceux où il n'y en a pas. Cependant, tous les langages de modélisation de processus ne permettent pas de définir qu'une unité de travail a une initialisation ou une finalisation. Une autre solution serait donc d'étendre le métamodèle de CA abstraits et de liaisons, afin que celui-ci permette de spécifier si une initialisation ou une finalisation d'unité de travail est manuelle. Cette extension pourrait prendre la forme d'une action spécifique qui serait une action manuelle. Cela permettrait que

les unités de travail ayant des initialisations et finalisations manuelles soient traitées différemment de celles n'en ayant pas.

CVL permet actuellement de dériver des modèles de processus résolus invalides, alors que le modèle de processus de base, le modèle d'abstraction de la variabilité (VAM), le modèle de réalisation de la variabilité (VRM) ainsi que le modèle de résolution de la variabilité (RM) sont valides. Pour rappel, nous avons identifié trois catégories d'erreurs qui rendent le modèle de processus résolu invalide. La première catégorie concerne les erreurs liées au non-respect de la compatibilité de type lorsqu'une nouvelle valeur est assignée à une instance de propriété. La deuxième catégorie englobe les erreurs liées au non-respect de la multiplicité d'une référence ou des contraintes définies sur le métamodèle du modèle de processus résolu. La troisième catégorie concerne les erreurs sémantiques, c'est-à-dire quand le modèle du processus résolu est bien conforme à son métamodèle, mais qu'il est trop permissif pour un métier donné ou qu'il contient des incohérences. Contraindre l'application des points de variation de CVL (opération à appliquer au modèle de base pour dériver un modèle résolu), à l'aide de contraintes exprimées de manière générique sur le métamodèle de CVL, permettrait d'éviter les erreurs de la première catégorie. Un outil générique d'analyse statique, qui assurerait que le modèle de processus résolu respecte bien les contraintes définies sur son métamodèle, permettrait d'éviter les erreurs de la deuxième catégorie. La modification du métamodèle de processus utilisé permettrait d'éviter les erreurs de la troisième catégorie. Cependant, dans les cas où cette modification s'avère trop coûteuse, un moteur de dérivation spécifique aux besoins d'un métier particulier permettrait de restreindre ce que le modèle de processus résolu peut capturer.

Un outil vérifiant la consistance des pré et post conditions associées aux CA, ainsi qu'aux unités de travail qu'ils automatisent, permettrait quant à lui de s'assurer qu'un modèle de CA abstraits et de liaisons est correct.

8.2.1.3 Suppression des redondances dans le modèle de CA abstraits et de liaisons

Dans le modèle de CA abstraits et de liaisons, il est actuellement nécessaire de spécifier quelle variante d'unité de travail chaque liste de CA automatise. Lorsque des variantes différentes d'une unité de travail sont automatisées par des listes différentes de CA, cela implique de définir plusieurs fois des éléments de modèle référençant la même unité de travail, en spécifiant à chaque fois à l'aide de conditions quelle est la variante de cette unité de travail. Afin de supprimer ces redondances, CVL pourrait être utilisé pour créer le modèle de CA abstraits et de liaisons correspondant à un modèle de processus résolu, en fonction des exigences d'un projet. Dans le modèle de CA abstraits et de liaisons, seule l'unité de travail automatisée par un CA serait spécifiée, sans préciser de quelle variante il s'agit. Une fois les exigences d'un projet définies, le modèle de CA abstraits et de liaisons serait transformé afin qu'il ne reste que les listes de CA correspondant aux variantes d'unités de travail qui sont dans le modèle de processus résolu.

8.2.2 Perspectives de recherche à court terme

En plus de l'application de notre approche à une famille de processus industriels (cf. section 7.2), une expérimentation, réalisée elle aussi dans un contexte industriel, permettrait de déterminer dans quelle proportion notre approche permet de réduire les temps de développement. Cette expérimentation pourrait consister à comparer le temps passé par un ensemble d'utilisateurs à réaliser un processus de développement logiciel manuellement et en utilisant l'approche et l'outillage de cette thèse. De plus, l'élaboration de l'approche proposée dans cette thèse ainsi que la conception et le développement de l'outillage support et des CA implique un investissement en terme de temps. Les résultats de l'expérimentation permettraient donc de déterminer combien de processus de développement logiciel il est nécessaire d'automatiser en suivant notre approche pour obtenir un retour sur investissement. Cette information serait utile pour déterminer dans quels cas il est rentable d'appliquer notre approche.

8.2.3 Perspectives de recherche à long terme

Nous abordons dans cette section les perspectives de recherche afin de gérer également la variabilité des processus qui a lieu pendant leur exécution et afin d'améliorer la réutilisation des CA.

8.2.3.1 Gestion de la variabilité des processus au moment de leur l'exécution

Un processus en cours d'exécution peut avoir besoin d'être adapté à cause de changements au niveau de son environnement d'exécution ou de l'organisation d'une entreprise, parce que certaines parties d'un processus dépendent de l'exécution de parties amont, ou encore à cause des spécificités d'un projet [BFG93]. Étendre l'approche actuellement proposée dans cette thèse afin qu'elle permette de mettre à jour automatiquement le modèle du processus en cours d'exécution en fonction de certains évènements permettrait de répondre à ce besoin. Les travaux de Morin [Mor10], qui permettent de reconfigurer dynamiquement des modèles en réponse aux changements dans leur environnement d'exécution, pourraient être réutilisés à cette fin. Il faudrait également que les différents artéfacts de développement liés au processus en cours d'exécution soient adaptés, tout en assurant leur cohérence [DO04]. Pour ce faire, une piste serait de réutiliser les travaux de Fouquet [Fou13], qui permettent de piloter à partir des modèles la reconfiguration dynamique de composants dans un environnement distribué.

La présence d'activités plus opportunes que celles définies par le processus peut justifier un besoin de déviation vis-à-vis de ce processus lors de son exécution [Vis90]. La difficulté est ici de dévier d'un processus tout en continuant à utiliser de manière cohérente les parties de ce processus qui restent pertinentes pour le projet [DO04]. Une piste de recherche serait donc d'étendre l'approche proposée dans cette thèse afin qu'elle permette de dévier du processus en cours d'exécution. À cette fin, les travaux de Kabbaj et al. [KLC08] pourraient être réutilisés. Ceux-ci permettent de détecter

les déviations vis-à-vis d'un processus au moment de son exécution et de gérer ces déviations en les acceptant ou en les refusant.

8.2.3.2 Amélioration de la réutilisation des CA

M4RAC ne s'appliquant qu'au niveau d'une famille de processus, elle ne permet pas d'améliorer la réutilisation de CA entre familles de processus. Étendre M4RAC, de manière à ce qu'elle s'applique à un ensemble de familles de processus, permettrait d'améliorer la réutilisation des CA entre familles de processus.

Nous avons identifié une première directive pour implémenter les CA afin d'améliorer leur réutilisation : lorsque la ligne de processus spécifie qu'un outil peut être substitué par un autre, alors, dans l'implémentation d'un CA, les parties qui dépendent de cet outil doivent être découplées des parties qui n'en dépendent pas. Une perspective de travail serait d'identifier des recommandations supplémentaires pour implémenter les CA, afin d'améliorer encore leur capacité à être réutilisés.

Il est difficile pour un humain d'identifier les contextes d'utilisation des CA, à cause de la définition en intention de processus, mais également à cause du nombre important de processus. En effet, il est dans ce cas difficile pour un humain de visualiser explicitement chacun des processus de la ligne et aussi de les intégrer mentalement. Un outil offrant un support à l'identification des contextes d'utilisation des CA serait une solution à ce problème. Il pourrait expliciter les différents processus d'une famille définie en intention, ainsi que les CA déjà associés à ces processus. Afin de limiter le nombre de processus affichés, il pourrait s'appuyer sur une définition de contextes d'utilisation de CA similaires, et donc n'afficher qu'un seul de ces contextes. La vérification de la consistance entre les pré et post-conditions associées aux CA permettrait quant à elle de détecter d'éventuelles erreurs au moment de l'association d'un CA à une unité de travail.

Appendices

Annexe A

Extrait des modèles et CA de la famille de processus de métamodélisation

Nous présentons dans cette annexe un extrait des modèles et des CA relatifs au cas d'application de la famille de processus de métamodélisation (cf. section 7.1).

Un extrait du VAM de cette famille est illustré en figure A.1. Les choix *tree editor*, *compiler*, *interpreter*, *checker*, *textual editor*, *EMFText*, *XText*, *version control system*, *SVN* et *Git* doivent être résolus manuellement. De plus, les choix *SVN* et *Git*, tout comme les choix *EMFText* et *XText*, sont mutuellement exclusifs. La résolution des autres choix est quant à elle dérivée. Seul le choix *version control system* est obligatoire (propriété *Is Implied By Parent* à vrai).

Dans le VRM, dont un extrait est présenté en figure A.2, les propriétés associées à la substitution de fin de lien *compiler creation into metamodeling activity* indiquent que la tâche de définition d'un compilateur est intégrée au processus le plus souvent utilisé (cf. figure 4.7) lorsque le choix *compiler* est sélectionné. Cette tâche est identifiée par le pointeur d'objet *compiler creation* et appartient initialement aux éléments de processus externes du modèle de processus de base (cf. figure 4.8). Les propriétés associées à la substitution de fin de lien *Fork into metamodeling activity* indiquent quant à elles que le nœud de parallélisation appartenant aux éléments de processus externes du modèle de processus de base est intégré au processus le plus souvent utilisé dès lors que des tâches sont exécutées en parallèle (choix *parallelization* résolu positivement). Le nœud de parallélisation est identifié par le pointeur d'objet *Fork* et le processus le plus souvent utilisé est représenté par l'activité *Metamodeling Activity*.

La figure A.3 illustre un extrait du modèle de liaisons de la famille de processus de métamodélisation. Les propriétés des pointeurs d'unités de travail (éléments de type `WorkUnitHandle` sur la figure) spécifient quelles sont les étapes (initialisation, exécution, finalisation) de ces unités de travail qui sont automatisées, et par quels CA. Ainsi, l'initialisation et la finalisation de la définition d'un métamodèle sont respectivement automatisées par un CA créant un projet EMF vide (*Primitive AC Create empty EMF*

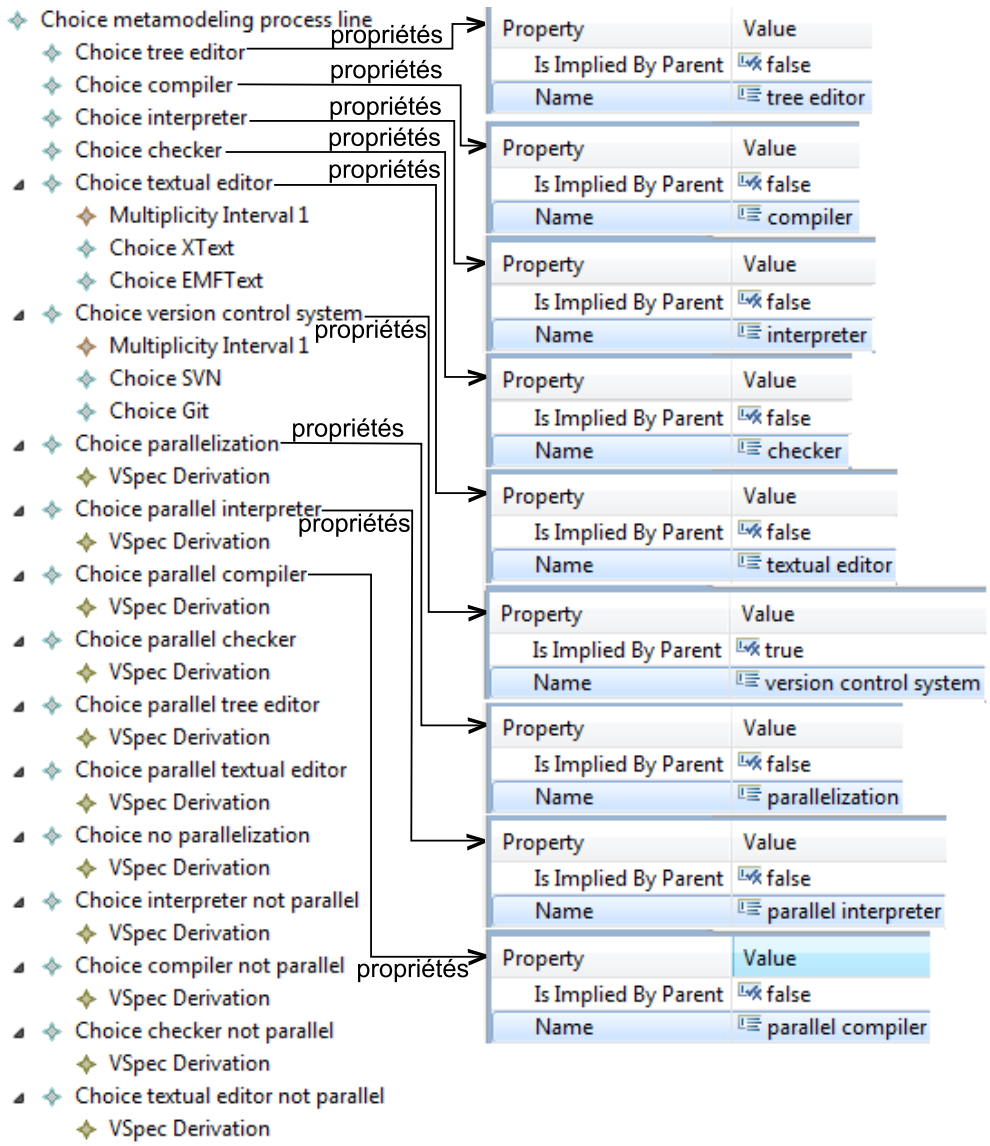


FIGURE A.1 – Extrait du VAM de la famille de processus de métamodélisation

project) et par un CA validant le métamodèle créé (*Primitive AC validate EMF model*). L'exécution de la définition d'un éditeur arborescent est automatisée par un CA générant cet éditeur (*Primitive AC generate tree editor*). Les initialisations des définitions d'un compilateur, d'un interpréteur et d'un vérificateur sont automatisées par des CA créant et initialisant un projet Kermeta (*Primitive AC create Kermeta compiler*, *Primitive AC create Kermeta interpreter*, *Primitive AC create Kermeta checker*). Enfin, la finalisation de la définition d'un interpréteur est automatisée par une liste de CA (*ACL put under version control*, constituée des CA *subclipse configuration*, *share project SVN* et *commit project SVN*) mettant cet interpréteur sous contrôle de version. Tous les CA sont im-

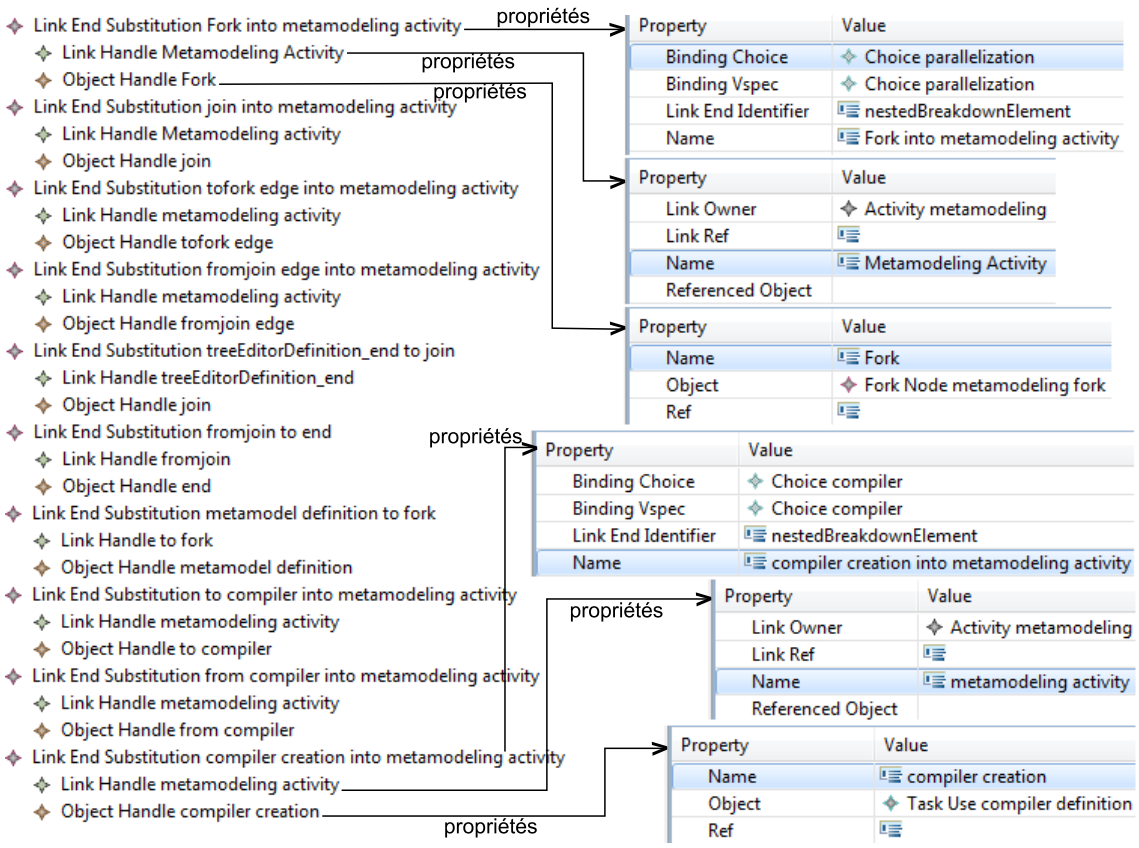


FIGURE A.2 – Extrait du VRM de la famille de processus de métamodélisation

plémentés par des actions Java Eclipse. Parmi elles, l'action *CreateEmptyEMFProject* est implémentée par la classe `CreateEmptyEMFProjectActivityAutomation`, comme illustré par la figure A.4. La méthode `run` de cette classe lance deux assistants Eclipse (cf. figure A.5). Le premier demande à l'acteur courant le nom du nouveau projet EMF, sauve cette information dans le modèle de contexte et crée le nouveau projet EMF. Le deuxième réalise la même séquence d'étapes concernant le fichier Ecore contenant le métamodèle.

Pour terminer, un extrait du RM est illustré par la figure A.6. Les propriétés des résolutions de choix *tree editor*, *SVN* et *Git* indiquent que les choix du même nom sont respectivement résolus à vrai, vrai et faux.

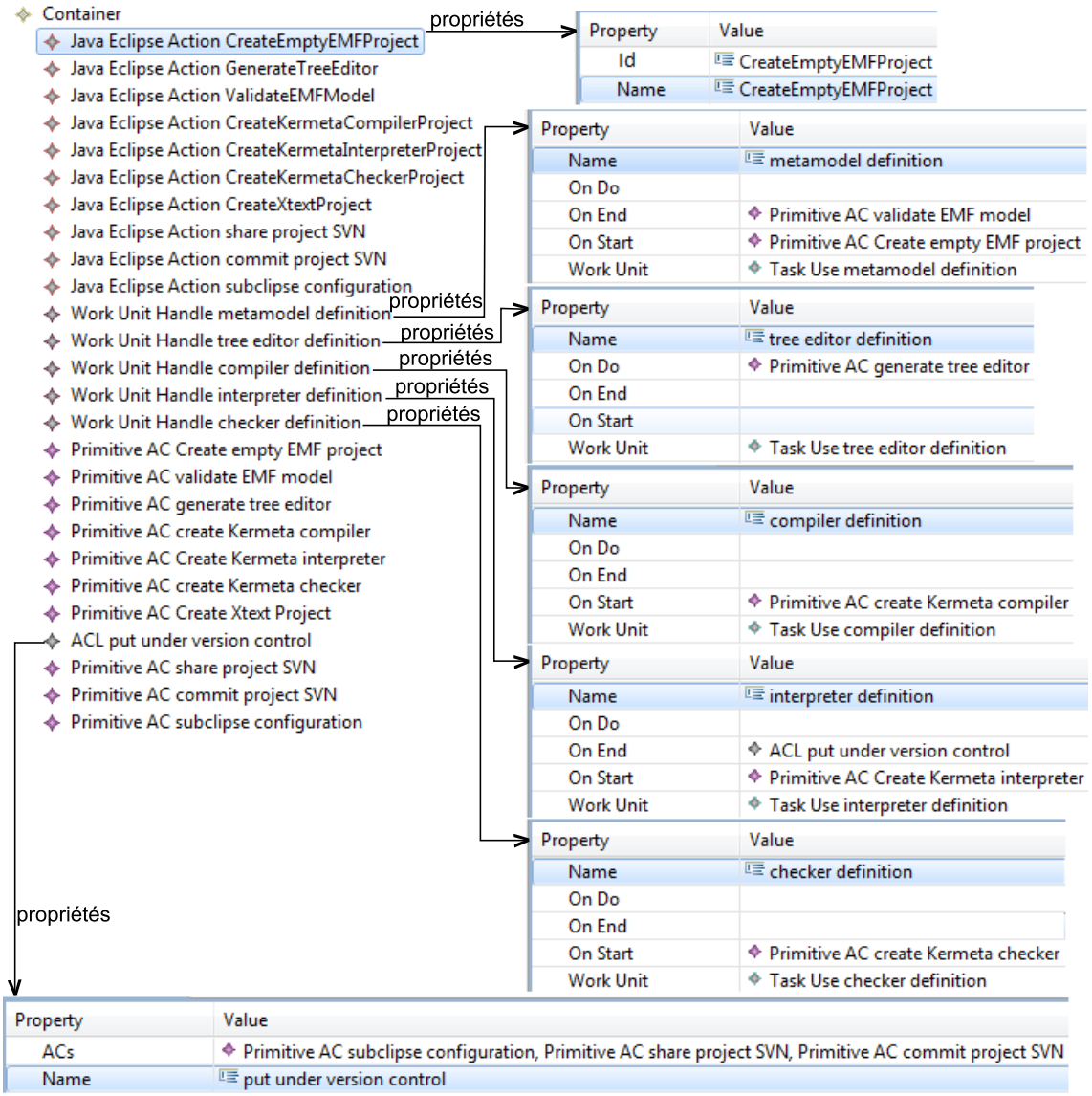


FIGURE A.3 – Extrait du modèle de liaisons de la famille de processus de métamodélisation

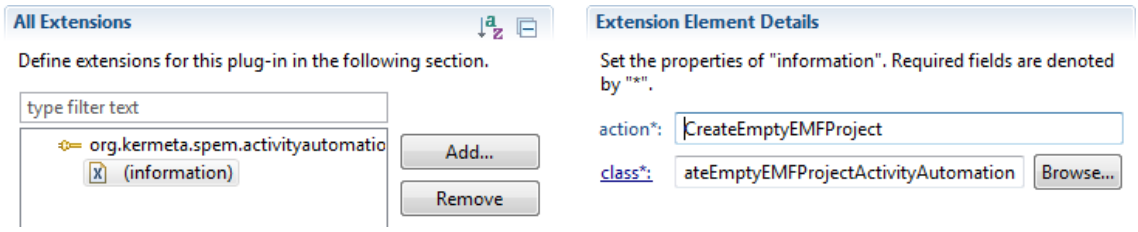


FIGURE A.4 – Déclaration de la classe implémentant l'action Java Eclipse *CreateEmptyEMFProject*

```
public class CreateEmptyEMFProjectActivityAutomation implements
        ActivityAutomation {

    private IContainer modelContainer;

    public CreateEmptyEMFProjectActivityAutomation() {
        // TODO Auto-generated constructor stub
    }

    @Override
    public void run(String contextModelPath) {
        try{
            EmptyProjectWizard2 emptyProjectWizard =
                    new EmptyProjectWizard2(this, contextModelPath);
            emptyProjectWizard.init(
                    PlatformUI.getWorkbench().getActiveWorkbenchWindow().getWorkbench(),
                    null);
            WizardDialog dialog =
                    new WizardDialog(
                            PlatformUI.getWorkbench().getActiveWorkbenchWindow().getShell(),
                            emptyProjectWizard);
            dialog.open();
        }catch(Exception e){
            e.printStackTrace();
        }

        StructuredSelection structuredSelection =
                new StructuredSelection(modelContainer);
        EcoreModelWizard2 ecoreModelWizard =
                new EcoreModelWizard2(contextModelPath);
        ecoreModelWizard.init(
                PlatformUI.getWorkbench().getActiveWorkbenchWindow().getWorkbench(),
                structuredSelection);
        WizardDialog dialogEcoreModelWizard =
                new WizardDialog(
                        PlatformUI.getWorkbench().getActiveWorkbenchWindow().getShell(),
                        ecoreModelWizard);
        dialogEcoreModelWizard.open();
    }

    public void setModelContainer(IContainer modelContainer){
        this.modelContainer = modelContainer;
    }
```

FIGURE A.5 – Code de la classe `CreateEmptyEMFProjectActivityAutomation`

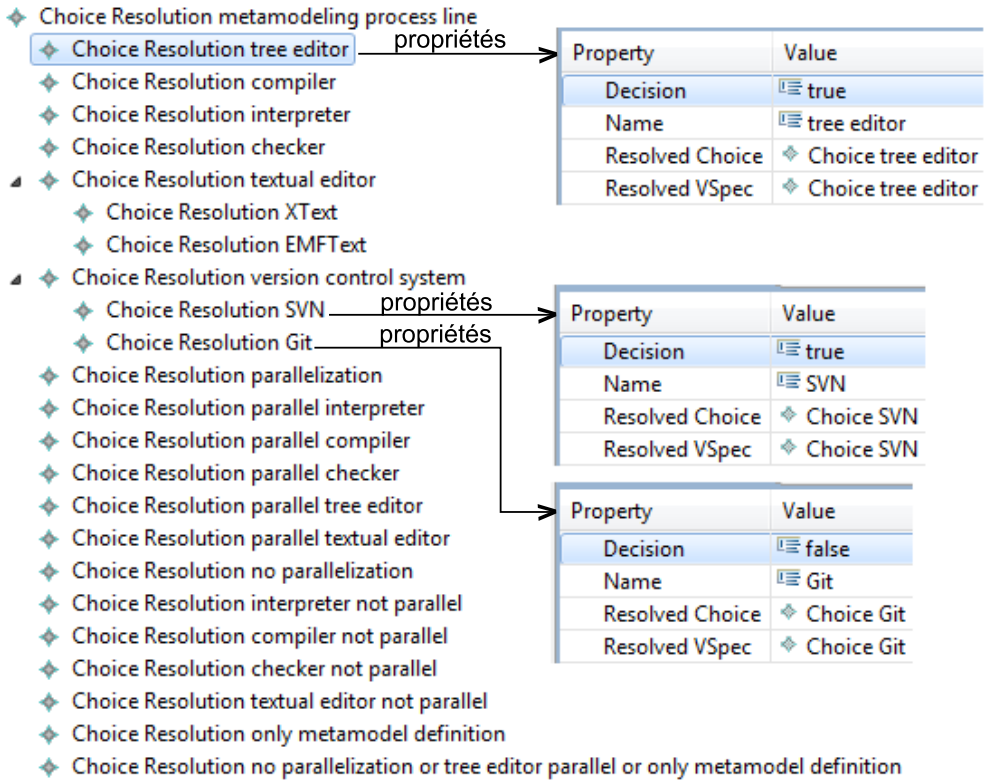


FIGURE A.6 – Extrait du RM du cas d'application de la famille de processus de métamodélisation de la section 7.1

Annexe B

Extrait des modèles et CA de la famille de processus de développement web Java

Nous présentons dans cette annexe un extrait des modèles et des CA relatifs au cas d'application de la famille de processus de développement web Java (cf. section 7.2).

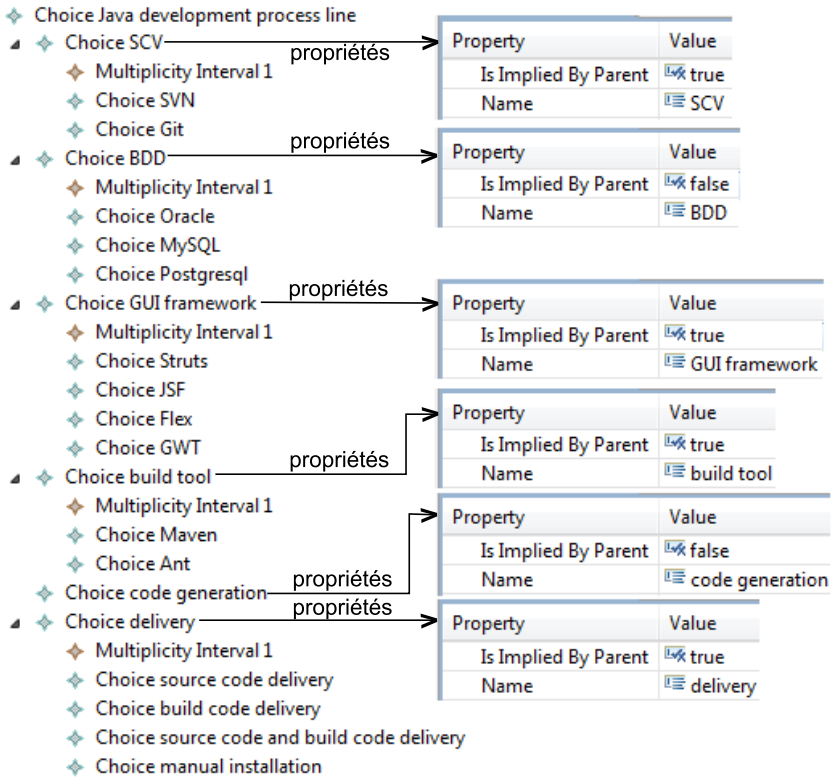


FIGURE B.1 – Extrait du VAM de la famille de processus de développement web Java

Le VAM de cette famille, dont un extrait est illustré par la figure B.1, comporte 22 choix. Leur résolution est manuelle et ils permettent de déterminer le SCV, la base de données, le *framework* pour l'IHM web, l'outil de compilation, si une étape de génération de code intervient avant le développement manuel, ainsi que le mode de livraison.

La figure B.2 illustre un extrait du VRM. La substitution d'objet *SVN replaced by Git* spécifie que l'outil SVN (référencé par le pointeur d'objet *SVN*) est remplacé par l'outil Git (référencé par le pointeur d'objet *Git*) lorsque le choix *Git* est sélectionné. L'existence d'objet *MySQL* spécifie que l'outil MySQL existe dans le modèle de processus résolu lorsque le choix *MySQL* est sélectionné. Sinon, cet outil est supprimé. La substitution d'objet *MySQL replaced by Oracle* spécifie quant à elle que l'outil MySQL est remplacé par l'outil *Oracle* lorsque le choix *Oracle* est sélectionné.

Un extrait du modèle de liaisons est illustré par la figure B.3. Le pointeur d'unité de travail *development* spécifie que l'initialisation de l'activité de développement (*Activity development*) est automatisée par la liste de CA *development initialization*. Celle-ci permet :

- d'installer les plug-ins WTP (*Primitive AC install wtp*) et Subclipse (*Primitive AC install subclipse*),
- de rédémarrer Eclipse (*Primitive AC restart*),
- d'initialiser un serveur Tomcat (*Primitive AC new Tomcat server*),
- de connecter un espace de travail Eclipse à un nouveau dépôt SVN (*Primitive AC subclipse configuration*)
- et de créer un nouveau dossier sur ce dépôt SVN (*Primitive AC create directory on remote repository*).

Les pointeurs d'unité de travail *project creation* et *first project creation* spécifient que les tâches de création de projets (*Task Use project creation* et *Task Use first project creation*) sont finalisées par une liste d'AC (*ACL share and commit project with SVN*) mettant ce projet sous contrôle de version (*Primitive AC share project with SVN*) et propageant son contenu sur un dépôt distant (*commit project with SVN*). Le pointeur d'unité de travail *projects modification* spécifie que la tâche de modification de projets (*Task Use projects modification*) est finalisée par un CA propageant les modifications apportées sur un dépôt distant (*Primitive AC commit the modified projects*). Comme pour le cas d'application précédent, tous les CA sont implémentés par des actions Java Eclipse. Parmi elles, l'action Java Eclipse *install wtp* a pour identifiant *installwtp*. Comme illustré par les figures B.4 et B.5, cette action est implémentée par la classe `InstallWTP` et nécessite comme information contextuelle le chemin de l'environnement Eclipse sur lequel installer le plug-in. La méthode `run` de cette classe lance l'exécution d'une commande Windows installant, sur un environnement Eclipse, les archives Java relatives au plug-in WTP (cf. figure B.6).

Enfin, la figure B.7 illustre un extrait du RM. Les propriétés des résolutions de choix *SCV*, *SVN*, *Git* et *code generation* indiquent que les choix du même nom sont respectivement résolus à vrai, vrai, faux et faux.

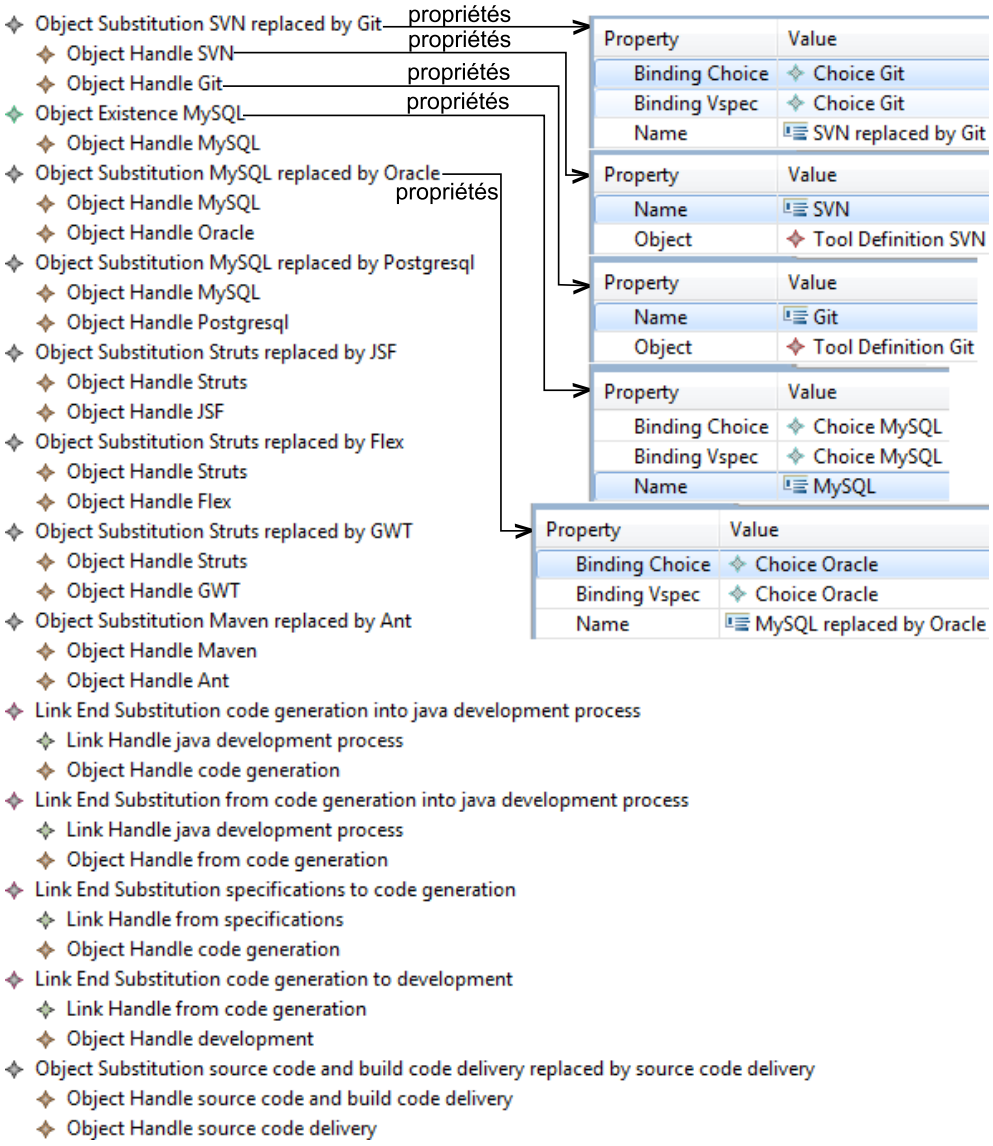


FIGURE B.2 – Extrait du VRM de la famille de processus de développement web Java

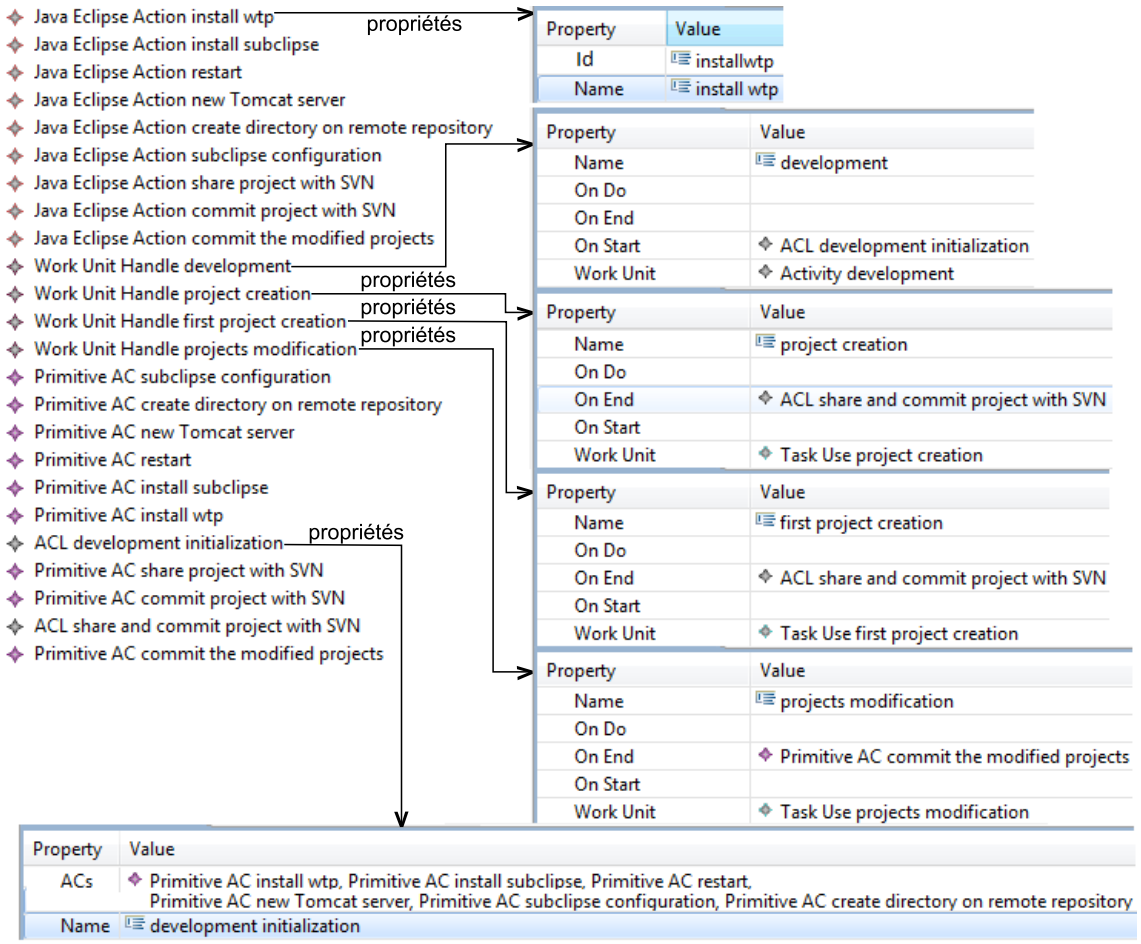


FIGURE B.3 – Extrait du modèle de liaisons de la famille de processus de développement web Java

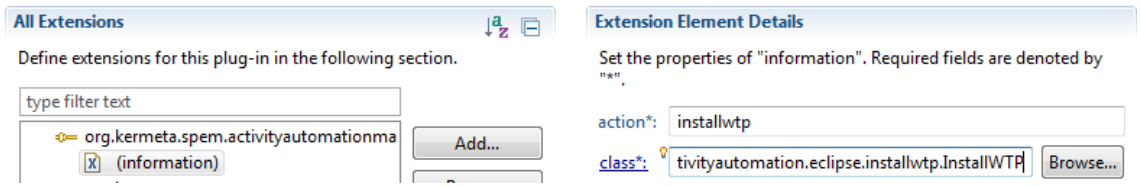


FIGURE B.4 – Déclaration de la classe implémentant l'action Java Eclipse *install wtp*

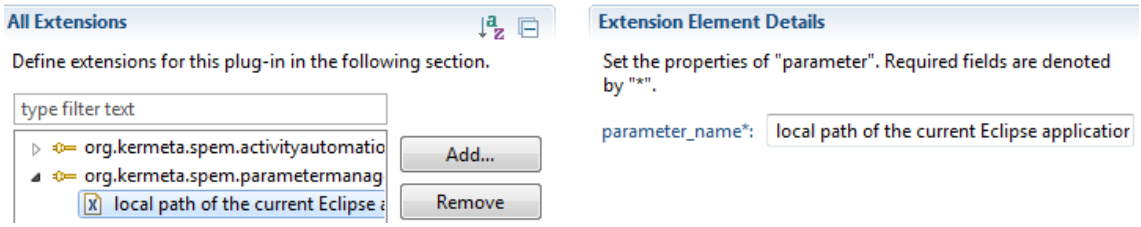


FIGURE B.5 – Information contextuelle du CA installant le plug-in WTP

```
public class InstallWTP implements ActivityAutomation {

    public InstallWTP() {
        // TODO Auto-generated constructor stub
    }

    @Override
    public void run(String contextModelPath) {
        ExecutionContext executionContext =
                ModelUtils.getExecutionContextRoot(
                        contextModelPath);
        String localPathOfTheCurrentEclipseApplication =
                ModelUtils.getValueOfKey(
                        "local path of the current Eclipse application",
                        executionContext);
        String localPathOfTheCurrentEclipseApplicationAfterConversion =
                StringUtils.convertPathIntoString(
                        localPathOfTheCurrentEclipseApplication);
        String cmd =
                localPathOfTheCurrentEclipseApplicationAfterConversion+
                " -application org.eclipse.equinox.p2.director"+
                " -repository http://download.eclipse.org/webtools/repository/indigo/," +
                "http://download.eclipse.org/datatools/updates," +
                "http://download.eclipse.org/graphiti/updates/0.9.2 " +
                "-installIU org.eclipse.graphiti.feature.feature.group/0.9.2.v20130211-0913,
        RunCommand.runCommand(cmd);
    }

}
```

FIGURE B.6 – Code de la classe `InstallWTP`

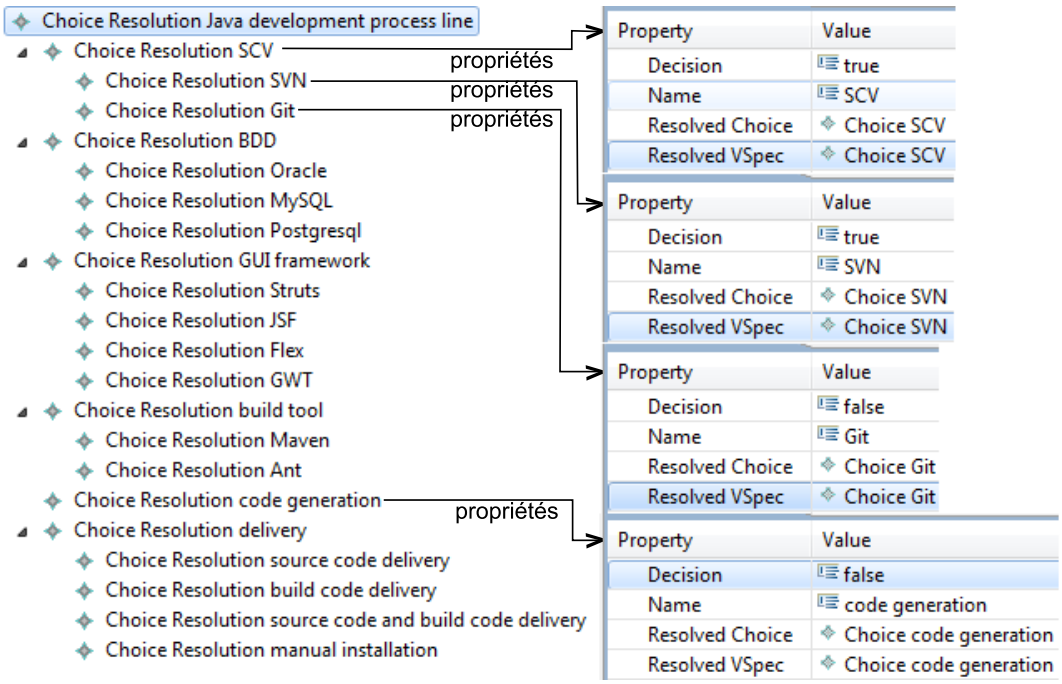


FIGURE B.7 – Extrait du RM du cas d'application de la famille de processus de développement web Java de la section 7.2

Bibliographie

[AB12] J.A.H. Alegría and M.C. Bastarrica. Building Software Process Lines with CASPER. In ICSSP, pages 170–179, 2012.

[ABC96] D. Avrilionis, N. Belkhatir, and P.-Y. Cunin. A unified framework for software process enactment and improvement. In ICSP, pages 102–111, 1996.

[ABM00] C. Atkinson, J. Bayer, and D. Muthig. Component-Based Product Line Development : The KobrA Approach. In SPLC, pages 289–309, 2000.

[ACG+12] M. Acher, P. Collet, A. Gaignard, P. Lahire, J. Montagnat, and R. France. Composing Multiple Variability Artifacts to Assemble Coherent Workflows. Software Quality Journal, 20(3-4) :689–734, 2012.

[AD91] L.C. Alexander and A.M. Davis. Criteria for Selecting Software Process Models. In COMPSAC, pages 521–528, 1991.

[Ald10] L. Aldin. Semantic Discovery and Reuse of Business Process Patterns. PhD thesis, School of Information Systems, Computing and Mathematics Brunel University, 2010.

[Amb99] S. W. Ambler. More Process Patterns Delivering Large-Scale Systems Using Object Technology. Cambridge University Press, 1999.

[ASKW11] A. Awad, S. Sakr, M. Kunze, and M. Weske. Design by Selection : A Reuse-Based Approach for Business Process Modeling. In ER, pages 332–345, 2011.

[BBM05] J. Bhuta, B. Boehm, and S. Meyers. Process Elements : Components of Software Process Architectures. In SPW, pages 332–346, 2005.

[BCCG07] R. Bendraou, B. Combemale, X. Cregut, and M.-P. Gervais. Definition of an Executable SPEM 2.0. In APSEC, pages 390–397, 2007.

[BDK07] J. Becker, P. Delfmann, and R. Knackstedt. Adaptive Reference Modeling : Integrating Configurative and Generic Adaptation Techniques for Information Models. In RM, pages 27–58, 2007.

[BDKK04] P. Becker, J. Delfmann, A. Dreiling, R. Knackstedt, and D. Kuropka. Configurative Process Modeling - Outlining an Approach to Increased Business Process Model Usability. In IRMA, pages 615–619, 2004.

[Ben07] R. Bendraou. UML4SPM : Un Langage De Modélisation De Procédés De Développement Logiciel Exécutable Et Orienté Modèle. PhD thesis, Université Pierre & Marie Curie - Paris VI, 2007.

[BFG93] S.C. Bandinelli, A. Fuggetta, and C. Ghezzi. Software Process Model Evolution in the SPADE Environment. IEEE TSE, 19(12) :1128–1144, 1993.

[BGB06] R. Bendraou, M.-P. Gervais, and X. Blanc. UML4SPM : An Executable Software Process Modeling Language Providing High-Level Abstractions. In EDOC, pages 297–306, 2006.

[BnCCG11] S. J. Bolaños Castro, R. G. Crespo, and V. H. M. García. Patterns of Software Development Process. IJIMAI, 1(4) :33–40, 2011.

[Boe88] B.W. Boehm. A Spiral Model of Software Development and Enhancement. Computer, 21(5) :61–72, 1988.

[Boo91] G. Booch. Object Oriented Design : With Applications. Benjamin/-Cummings Publishing Company, 1991.

[BSE03] F. Budinsky, D. Steinberg, and R. Ellersick. Eclipse Modeling Framework : A Developer's Guide. Addison-Wesley Professional, 2003.

[CA05] K. Czarnecki and M. Antkiewicz. Mapping Features to Models : A Template Approach Based on Superimposed Variants. In GPCE, pages 422–437, 2005.

[CB05] S. Clarke and E. Baniassad. Aspect-Oriented Analysis and Design : The Theme Approach. Addison-Wesley, 2005.

[CBHB07] M. Cataldo, M. Bass, J.D. Herbsleb, and L. Bass. On Coordination Mechanisms in Global Software Development. In ICGSE, pages 71–80, 2007.

[CC07] D. Ciuksys and A. Caplinskas. Reusing Ontological Knowledge about Business Processes in IS Engineering : Process Configuration Problem. Informatica, 18(4) :585–602, 2007.

[CDCC+13] E. Céret, S. Dupuy-Chessa, G. Calvary, A. Front, and D. Rieu. A Taxonomy of Design Methods Process Models. Information and Software Technology, 55(5) :795 – 821, 2013.

[CKO92] B. Curtis, M. I. Kellner, and J. Over. Process Modeling. Commun. ACM, 35(9) :75–90, 1992.

[CLM+99] A. G. Cass, B. S. Lerner, E. K. McCall, L. J. Osterweil, and A. Wise. Logically Central, Physically Distributed Control in a Process Runtime Environment. Technical Report 99-65, University of Massachusetts at Amherst, 1999.

[CLS+00] A. G. Cass, B. S. Lerner, S. M. Sutton, Jr., E. K. McCall, A. Wise, and L. J. Osterweil. Little-JIL/Juliette : A Process Definition Language and Interpreter. In ICSE, pages 754–757, 2000.

[CN01] P. Clements and L. Northrop. Software Product Lines : Practices and Patterns. Addison-Wesley Longman Publishing Co., Inc., 2001.

[Cox85] B.J. Cox. Object Oriented Programming. Addison-Wesley,Reading, MA, 1985.

[Crn02] I. Crnkovic. Building Reliable Component-Based Software Systems. Artech House, Inc., 2002.

[DB10] W. Derguech and S. Bhiri. Reuse-Oriented Business Process Modelling Based on a Hierarchical Structure. In BPM, pages 301–313, 2010.

[Dij82] E. W. Dijkstra. On the Role of Scientific Thought. In Selected Writings on Computing : A Personal Perspective, pages 60–66. Springer-Verlag, 1982.

[Dio93] R. Dion. Process Improvement and the Corporate Balance Sheet. IEEE Software, 10(4) :28–35, 1993.

[DO04] J.-C. Derniame and F. Oquendo. Key Issues and New Challenges in Software Process Technology. UPGrade, The European Journal for the Informatics Professional, 5(5) :11–16, 2004.

[FHMP+11] F. Fleurey, Ø. Haugen, B. Møller-Pedersen, A. Svendsen, and X. Zhang. Standardizing Variability - Challenges and Solutions. In SDL Forum, pages 233–246, 2011.

[FL03] P. Fettke and P. Loos. Classification of reference models : a methodology and its application. Information Systems and e-Business Management, 1(1) :35–53, 2003.

[Fou13] F. Fouquet. Kevoree : Model@Runtime pour le développement continu de systèmes adaptatifs distribués hétérogènes. PhD thesis, University of Rennes 1, 2013.

[FPLPdL01] S. T. Fiorini, J. C. S. Prado Leite, and C. J. Pereira de Lucena. Process Reuse Architecture. In CAiSE, pages 284–298, 2001.

[Fra99] U. Frank. Conceptual Modelling as the Core of the Information Systems Discipline – Perspectives and Epistemological Challenges. In AMCIS, pages 695–697, 1999.

[GLKD98] K. Gary, T. Lindquist, H. Koehnemann, and J.-C. Derniame. Component-based Software Process Support. In ASE, pages 196–199, 1998.

[GM05] R. J. Glushko and T. S. McGrath. Document Engineering : Analyzing And Designing Documents For Business Informatics & Web Services. MIT Press, 2005.

[Guy13] C. Guy. Facilités de typage pour l'ingénierie des langages. PhD thesis, UniversitÃ© de Rennes 1, 2013.

[GvdAJV07] F. Gottschalk, W. M. P. van der Aalst, and M. H. Jansen-Vullers. SAP WebFlow Made Configurable : Unifying Workflow Templates into a Configurable Model. In BPM, pages 262–270, 2007.

[GvdAJVLR08] F. Gottschalk, W. M.P. van der Aalst, M. H. Jansen-Vullers, and M. La Rosa. Configurable Workflow Models. Cooperative Information Systems, 17(2) :177–221, 2008.

[HABQO11] J.A. Hurtado Alegría, M.C. Bastarrica, A. Quispe, and S.F. Ochoa. An MDE Approach to Software Process Tailoring. In ICSSP, pages 43–52, 2011.

[Ham96] Michael Hammer. Beyond Reengineering : How the Process-Centered Organization is Changing Our Work and Our Lives. Harper Collins Publishers, 1996.

[HBR10] A. Hallerbach, T. Bauer, and M. Reichert. Capturing Variability in Business Process Models : the Provop Approach. Software Maintenance, 22(67) :519–546, 2010.

[HIMT96] N. Hanakawa, H. Iida, K.-i. Matsumoto, and K. Torii. A Framework of Generating Software Process Including Milestones for Object-Oriented Development Method. In APSEC, pages 120–130, 1996.

[HMR06] J. D. Herbsleb, A. Mockus, and J. A. Roberts. Collaboration in Software Engineering Projects : A Theory of Coordination. In ICIS, 2006.

[Hum88] W. S. Humphrey. The Software Engineering Process : Definition and Scope. SIGSOFT Softw. Eng. Notes, 14(4) :82–83, 1988.

[HWRK11] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen. Empirical Assessment of MDE in Industry. In ICSE, pages 471–480, 2011.

[HZG+97] J. Herbsleb, D. Zubrow, D. Goldenson, W. Hayes, and M. Paulk. Software Quality and the Capability Maturity Model. Commun. ACM, 40(6) :30–40, 1997.

[IMCH+07] C. Iochpe, C. Ming Chiao, G. Hess, G. Nascimento, L. Thom, and M. Reichert. Towards an intelligent workflow designer based on the reuse of workflow patterns. In WBPM, 2007.

[Ins95] Software Engineering Institute. The Capability Maturity Model : Guidelines for Improving the Software Process. Addison-Wesley, 1995.

[Ins13] CMMI Institute. Maturity Profile Reports, September 2013.

[ISO07] ISO/IEC. ISO/IEC 24744 :2007 Software Engineering – Metamodel for Development Methodologies, 2007.

[Ist13] P. Istoan. Methodology for the derivation of product behaviour in a Software Product Line. PhD thesis, University of Rennes 1 and University of Luxembourg, 2013.

[JCV12] J.-M. Jézéquel, B. Combemale, and D. Vojtisek. Ingénierie Dirigée par les Modèles : des concepts à la pratique... Ellipses, 2012.

[Jéz12] J.-M. Jézéquel. Model-Driven Engineering for Software Product Lines. ISRN Software Engineering, 2012(2012), 2012.

[JMD04] T. Javed, M. Maqsood, and Q. S. Durrani. A Study to Investigate the Impact of Requirements Instability on Software Defects. SIGSOFT Softw. Eng. Notes, 29(3) :1–7, 2004.

[KB10] V. Kulkarni and S. Barat. Business Process Families Using Model-Driven Techniques. In BPM, pages 314–325, 2010.

[KCH+90] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, Carnegie-Mellon University Soft. Eng. Institute, 1990.

[KKL+98] C. K. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh. FORM : A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. Annals of Software Engineering, 5 :143–168, 1998.

[KL07] B. Korherr and B. List. A UML 2 Profile for Variability Models and their Dependency to Business Processes. In DEXA, pages 829–834, 2007.

[KLC08] M. Kabbaj, R. Lbath, and B. Coulette. A Deviation Management System for Handling Software Process Enactment Evolution. In ICSP, pages 186–197, 2008.

[KR10] M. Khaari and R. Ramsin. Process Patterns for Aspect-Oriented Software Development. In ECBS, pages 241–250, 2010.

[Kru92] C. W. Krueger. Software Reuse. ACM Comput. Surv., 24(2) :131–183, 1992.

[Kru06] C. W. Krueger. Introduction to the Emerging Practice of Software Product Line Development. Methods and Tools, 14(3) :3–15, 2006.

[KSG04] J. Kim, M. Spraragen, and Y. Gil. An Intelligent Assistant for Interactive Workflow Composition. In IUI, pages 125–131, 2004.

[KY12] A. Kumar and W. Yao. Design and management of flexible process variants using templates and rules. Comput. Ind., 63(2) :112–130, 2012.

[KYSR09] E. Kouroshfar, H. Yaghoubi Shahir, and R. Ramsin. Process Patterns for Component-Based Software Development. In CBSE, pages 54–68, 2009.

[LK04] K. Lee and K. Kang. Feature Dependency Analysis for Product Line Component Design. In ICSR, pages 69–85, 2004.

[LKCC00] K. Lee, K. C. Kang, W. Chae, and B. W. Choi. Featured-Based Approach to Object-Oriented Engineering of Applications for Reuse. Softw. Pract. Exper., 30(9) :1025–1046, 2000.

[LRDtHM11] M. La Rosa, M. Dumas, A. H.M. ter Hofstede, and J. Mendling. Configurable Multi-Perspective Business Process Models. Information Systems : Databases : Their Creation, Management and Utilization, 36(2) :313–340, 2011.

[LRvdADM13] M. La Rosa, W. M.P. van der Aalst, M. Dumas, and F. P. Milani. Business Process Variability Modeling : A Survey. ACM Computing Surveys, 2013.

[LS07] R. Lu and S. Sadiq. On the Discovery of Preferred Work Practice Through Business Process Variants. In ER, pages 165–180, 2007.

[Mar03] R. C. Martin. Agile Software Development : Principles, Patterns, and Practices. Prentice Hall/Pearson Education, 2003.

[MBF13] S. Mosser and M. Blay-Fornarino. "Adore", a Logical Meta-model Supporting Business Process Evolution. Sci. Comput. Program., 78(8) :1035–1054, 2013.

[Mee10] S. Meerkamm. Configuration of Multi-perspectives Variants. In BPM, pages 277–288, 2010.

[MHY08] M. Moon, M. Hong, and K. Yeom. Two-Level Variability Analysis for Business Process with Reusability and Extensibility. In COMPSAC, pages 263–270, 2008.

[Mor10] B. Morin. Leveraging Models from Design-time to Runtime to Support Dynamic Variability. PhD thesis, University of Rennes 1, 2010.

[MRGP08] T. Martínez-Ruiz, F. García, and M. Piattini. Towards a SPEM v2.0 Extension to Define Process Lines Variability Mechanisms. In SERA, pages 115–130, 2008.

[MRGPM11] T. Martínez-Ruiz, F. García, M. Piattini, and J. Münch. Modelling Software Process Variability : an Empirical Study. IET Software, 5(2) :172–187, 2011.

[NCH11] T. Nguyen, A. Colman, and J. Han. Modeling and Managing Variability in Process-Based Service Compositions. In ICSOC, pages 404–420, 2011.

[Nor08] L. Northrop. Software Product Lines Essentials, 2008.

[OAS07] OASIS. Web Services Business Process Execution Language Version 2.0, 2007.

[oD11] United States Department of Defense. Technology Readiness Assessment (TRA) Guidance, 2011.

[OMG06] OMG. Meta Object Facility (MOF) 2.0 Core Specification, 2006.

[OMG08] OMG. Software and Systems Process Engineering Metamodel Specification (SPEM) Version 2, 2008.

[OMG10] OMG. Documents associated with Object Constraint Language, Version 2.2, 2010.

[OMG11a] OMG. Documents Associated with Business Process Model and Notation (BPMN) Version 2.0, 2011.

[OMG11b] OMG. OMG Unified Modeling Language (OMG UML), Infrastructure Version 2.4.1, 2011.

[OMG11c] OMG. OMG Unified Modeling Language (OMG UML), Superstructure Version 2.4.1, 2011.

[OMG12] OMG. Diagram Definition (DD) Version 1.0, 2012.

[Ost87] L. Osterweil. Software Processes are Software Too. In ICSE, pages 2–13, 1987.

[PBL05] K. Pohl, G. Böckle, and F. J. van der Linden. Software Product Line Engineering : Foundations, Principles and Techniques. Springer, 2005.

[PEM95] G. Pérez, K. Emam, and N. H. Madhavji. Customising Software Process Models. In SPT, pages 70–78, 1995.

[PS05] V. Pankratius and W. Stucky. A Formal Foundation for Workflow Composition, Workflow View Definition, and Workflow Normalization Based on Petri Nets. In APCCM, pages 79–88, 2005.

[PSWW05] F. Puhlmann, A. Schnieders, J. Weiland, and M. Weske. Variability Mechanisms for Process Models. Technical report, PESOA-Report TR 17/2005, Process Family Engineering in Service-Oriented Applications (PESOA), 2005.

[RBJ07] R. Ramos, O. Barais, and J.M. Jézéquel. Matching Model-Snippets. In MoDELS 07, pages 121–135, 2007.

[RBSS09] I. Reinhartz-Berger, P. Soffer, and A. Sturm. Organisational reference models : Supporting an adequate design of local business processes. International Journal of Business Process Integration and Management, 4(2) :134–149, 2009.

[RBSS10] I. Reinhartz-Berger, P. Soffer, and A. Sturm. Extending the Adaptability of Reference Models. IEEE Transactions on Systems, Man and Cybernetics, Part A : Systems and Humans, 40(5) :1045–1056, 2010.

[RCB+11] E. Rouillé, B. Combemale, O. Barais, D. Touzet, and J.-M. Jézéquel. Bridging the Gap between Software Process and Software Development. In IDM, pages 103–108, 2011.

[RCB+12] E. Rouillé, B. Combemale, O. Barais, D. Touzet, and J.-M. Jézéquel. Leveraging CVL to Manage Variability in Software Process Lines. In APSEC, pages 148–157, 2012.

[RCB+13a] E. Rouillé, B. Combemale, O. Barais, D. Touzet, and J.-M. Jézéquel. Improving Reusability in Software Process Lines. In SEAA, pages 90–94, 2013.

[RCB+13b] E. Rouillé, B. Combemale, O. Barais, D. Touzet, and J.-M. Jézéquel. Integrating Software Process Reuse and Automation. In APSEC, pages 380–387, 2013.

[RDH+08] M. Rosa, M. Dumas, A. H. Hofstede, J. Mendling, and F. Gottschalk. Beyond Control-Flow : Extending Business Process Configuration to Roles and Objects. In ER, pages 199–215, 2008.

[RGC04] F. Ruiz-Gonzalez and G. Canfora. Software Process : Characteristics, Technology and Environments. UPGrade, The European Journal for the Informatics Professional, 5(5) :6–10, 2004.

[RHEdA05] N. Russell, A. H. M. Hofstede, D. Edmond, and W. M. P. der Aalst. Workflow Data Patterns : Identification, Representation and Tool Support. In ER, pages 353–368, 2005.

[RK08] M. Razavian and R. Khosravi. Modeling Variability in Business Process Models Using UML. In ITNG, pages 82–87, 2008.

[RMvdT09] H. A. Reijers, R. S. Mans, and R. A. van der Toorn. Improved Model Management with Aggregated Business Process Models. Data Knowledge Engineering, 168(2) :221–243, 2009.

[Rom05] H. D. Rombach. Integrated Software Process and Product Lines. In ISPW, pages 83–90, 2005.

[Roy87] W. W. Royce. Managing the Development of Large Software Systems : Concepts and Techniques. In ICSE, pages 328–338, 1987.

[RTM10] S. H. Ripon, K. H. Talukder, and M. K. I. Molla. Modelling Variability for System Families. Malaysian Journal of Computer Science, 16(1) :37–46, 2010.

[RvdA07] M. Rosemann and W. M. P. van der Aalst. A Configurable Reference Modelling Language. Information Systems, 32(1) :1–23, 2007.

[SA09] A. Sadovykh and A. Abherve. MDE Project Execution Support via SPEM Process Enactment. In MDTPI, 2009.

[SB11] J. Simmonds and M. C. Bastarrica. Modeling Variability in Software Process Lines. Technical report, Universidad de Chile, 2011.

[SBPM09] D. Steinberg, F. Budinsky, M Paternostro, and E. Merks. EMF : Eclipse Modeling Framework 2.0. Addison-Wesley Professional, 2009.

[Sch00] A.-W. Scheer. Aris-Business Process Modeling. Springer-Verlag New York, Inc., 2000.

[SCO07] B. I. Simidchieva, L. A. Clarke, and L. J. Osterweil. Representing Process Variation with a Process Family. In ICSP, pages 109–120, 2007.

[SCS10] E. Santos, J. Castro, and O. Sánchez, J.and Pastor. A Goal-Oriented Approach for Variability in BPMN. In WER, pages 17–28, 2010.

[SdSSB+10] I. Sharon, M. dos Santos Soares, J. Barjis, J. van den Berg, and J. L. M. Vrancken. A Decision Framework for Selecting a Suitable Software Development Process. In ICEIS, pages 34–43, 2010.

[SO98] X. Song and L. J. Osterweil. Engineering Software Design Processes to Guide Process Execution. IEEE Transactions on Software Engineering, 24(9) :759–775, 1998.

[Ter09] T. Ternité. Process Lines : A Product Line Approach Designed for Process Model Development. In SEAA, pages 173–180, 2009.

[Tho05] O. Thomas. Understanding the Term Reference Model in Information Systems Research : History, Literature Analysis and Explanation. In BPM, pages 484–496, 2005.

[TI93] T. Tamai and A. Itou. Requirements and Design Change in Large-Scale Software Development : Analysis from the Viewpoint of Process Backtracking. In ICSE, pages 167–176, 1993.

[VDATHKB03] W. M. P. Van Der Aalst, A. H. M. Ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow Patterns. Distrib. Parallel Databases, 14(1) :5–51, 2003.

[VG07] M. Voelter and I. Groher. Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. In SPLC, pages 233–242, 2007.

[Vis90] W. Visser. More or Less Following a Plan During Design : Opportunistic Deviations in Specification. International Journal of Man-Machine Studies, 33(3) :247–278, 1990.

[(WF99] WorkFlow Managment Coalition (WFMC). Terminology & Glossary, Document Number WFMC-TC-1011, 1999.

[Wis06] A. Wise. Little-jil 1.5 language report. Technical Report UM-CS-2006-51, Department of Computer Science, University of Massachusetts, Amherst, MA, 2006.

[WKK+11] M. Weidmann, F. Koetter, M. Kintz, D. Schleicher, and R. Mietzner. Adaptive Business Process Modeling in the Internet of Services (ABIS). In ICIW, pages 29–34, 2011.

[WL99] D. M. Weiss and C. T. R. Lai. Software Product-Line Engineering : A Family-Based Software Development Process. Addison-Wesley Professional, 1999.

[YLL+08] Y. Yu, A. Lapouchnian, S. Liaskos, J. Mylopoulos, and J. C.S.P. Leite. From Goals to High-Variability Software Design. In ISMIS, pages 1–16, 2008.

[Zam01] K. Z. Zamli. Process Modeling Languages : A Literature Review. Malaysian Journal of Computer Science, 14(2) :26–37, 2001.

[Zhu03] H. Zhuge. Component-based workflow systems development. Decision Support Systems, 35(4) :517 – 536, 2003.

[ZXD05] W. Zhongjie, X. Xiaofei, and Z. Dechen. A Component Optimization Design Method Based on Variation Point Decomposition. In SERA, pages 399–406, 2005.

Liste des figures

Liste des tables

Liste des publications

[RCB+11] E. Rouillé, B. Combemale, O. Barais, D. Touzet, and J.-M. Jézéquel. Bridging the Gap between Software Process and Software Development. In IDM, pages 103–108, 2011.

[RCB+12] E. Rouillé, B. Combemale, O. Barais, D. Touzet, and J.-M. Jézéquel. Leveraging CVL to Manage Variability in Software Process Lines. In APSEC, pages 148–157, 2012.

[RCB+13a] E. Rouillé, B. Combemale, O. Barais, D. Touzet, and J.-M. Jézéquel. Improving Reusability in Software Process Lines. In SEAA, pages 90–94, 2013.

[RCB+13b] E. Rouillé, B. Combemale, O. Barais, D. Touzet, and J.-M. Jézéquel. Integrating Software Process Reuse and Automation. In APSEC, pages 380–387, 2013.

Résumé

De nombreux outils existent afin de faire face à la complexité des logiciels et des projets de développement logiciel. Leur utilisation est cependant à l'origine de tâches manuelles répétitives, sources d'erreurs et coûteuses en temps.

L'automatisation de ces tâches permet de gagner en productivité. Mais la difficulté est de déterminer quand une automatisation de tâche manuelle répétitive doit être réutilisée, ainsi que de créer des automatisations réutilisables à travers leurs différents cas d'utilisation.

Nous proposons donc une approche outillée pilotant la réutilisation des automatisations de tâches manuelles répétitives par les processus de développement logiciel, où un processus de développement logiciel décrit les étapes à réaliser pour mener à bien un projet de développement logiciel. Cette approche consiste à capitaliser sur un ensemble de processus et à réutiliser des processus de cet ensemble en fonction des exigences des projets, indépendamment du formalisme utilisé pour définir les processus. Des automatisations de tâches manuelles répétitives sont liées aux étapes des processus qu'elles automatisent. Ce lien permet de savoir quelles automatisations utiliser pour un projet donné et quand. Il permet également d'expliciter les différents cas d'utilisation de chaque automatisation. Cette information est utilisée afin de créer des automatisations réutilisables à travers leurs différents cas d'utilisation.

L'approche ainsi que l'outillage associé ont été appliqués sur une famille de processus industriels de développement Java ainsi que sur une famille de processus consistant à définir et outiller un langage de modélisation.

Abstract

Many tools have been developped in order to manage the complexity of the software and of the software development projects. However, using these tools is the source of manual recurrent tasks that are error prone and time consuming.

Automating these tasks enables to improve the productivity. But the difficulties are i) to determine when the automation of a manual recurrent task must be used, and ii) to create automations that are reusable across their different use cases.

We propose a tool-supported approach that drives the reuse of the automations of the manual recurrent tasks by software processes. A software process defines the sequence of steps to perform in order to realize a software engineering project. This approche consists of capitalizing on a set of software processes and of reusing processes from this set according to projects' requirements and independently of the formalism used to define the processes. The automations of the manual recurrent tasks are bound to the processes' steps they automate. This binding enables to know which automations to reuse for a specific project and when to reuse these automations during the project. This binding also enables to explicit the different use cases of each automation. We use this information to create automations that are reusable across their different use cases.

We applied this tool-supported approach on a family of Java development processes coming from the industry as well as on a family of processes consisting of designing and implementing a modeling language.